[redacted]

LEVEL IV

TECHNICAL REPORT #7

TECHNIQUES FOR OPERATING SYSTEM MACHINES

July 1977

[redacted]

DISCLAIMERS

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The citation of trade names and names of manufacturers in this report is not to be construed as official Government endorsement or approval of commercial products or services referenced herein.

DISPOSITION

Destroy this report when it is no longer needed. Do not return it to the originator.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

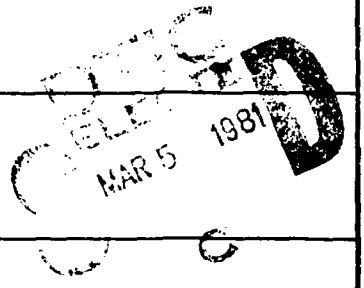
READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: TR-7	2. GOVT ACCESSION NO.: AD-A095989
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE *(and Subtitle)*: Techniques for Operating System Machines.	5. TYPE OF REPORT & PERIOD COVERED: Final Report for period Oct. 1976 — May 1977.
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Higher Order Software, Inc. Margaret Hamilton, Susan Sayder	8. CONTRACT OR GRANT NUMBER(s): DAAG29-76-C-0061
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Higher Order Software, Inc. Cambridge, MA 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: Higher Order Software, Inc. Cambridge, MA 02139	12. REPORT DATE: July 1977
	13. NUMBER OF PAGES: 132
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Army Research Office Research Triangle Park, NC	15. SECURITY CLASS. *(of this report)*: Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*



18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Operating Systems, Machines, Formal Methodology, Axioms, Control Strutures, Data Types, Resource Allocation, Security, System Layers.

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

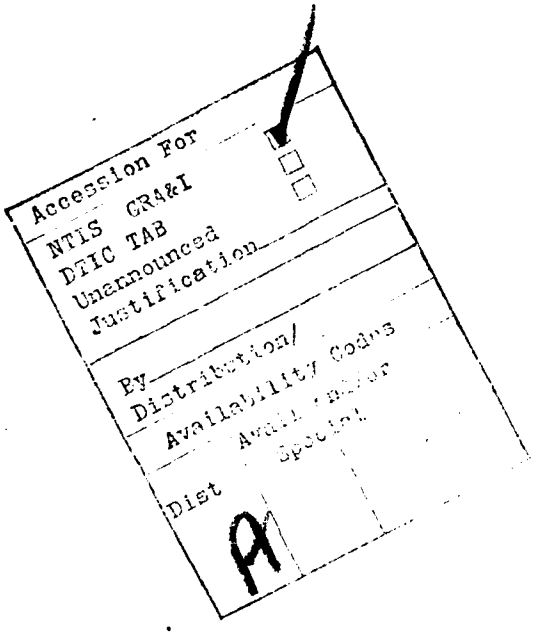
ABSTRACT: One of the most difficult problems facing system designers is that of resource allocation to a particular machine. In this report we discuss the basic concepts of resource allocation to, by and from an operating system and the techniques of describing these concepts in terms of the HOS specification language, AXES. We show here that Higher Order Software (HOS) systems (including operating systems) are secure. The HOS/AXES concepts are demonstrated by example specifications of an operating system which uses as a model the APOLLO

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

20. on-board flight software system. The concept of a Higher Order Machine (HOM), whose purpose is to replace today's intermediate machine layers and communicate directly to an HOS specified system is discussed.



HIGHER ORDER SOFTWARE, INC.
843 Massachusetts Avenue
Cambridge, MA 02139

TECHNICAL REPORT #7

TECHNIQUES FOR OPERATING SYSTEM MACHINES

July 1977

Prepared for
Army Research Office
Research Triangle Park, NC

ACKNOWLEDGEMENTS

This report was prepared under Contract No. DAAG29-76-C-0061, sponsored by the Department of the Army, U.S. Army Research Office, Research Triangle Park, North Carolina.

The authors would like to express appreciation to Andrea Davis and Gail Lopes for the preparation of this report.

TABLE OF CONTENTS

Section I

OPERATING SYSTEM MACHINES WITH RESPECT TO RESOURCE ALLOCATION

by

M. Hamilton and S. Zeldin

TABLE OF CONTENTS

FIGURES

1.0 INTRODUCTION

An operating system enables a user to share resources of a computer system [1]. General characteristics of such a system include scheduling computations, protecting processes from interfering with one another, accessing instructions and data, and measuring performance of the system. On one hand, an operating system must interface with the user, while, on the other hand, the operating system must interface with the computer itself. Changing user requirements often requires extensive modifications to such systems. The specification for such a system must be designed so that such modifications do not imply a redesign. In addition, we must try to separate those functions that directly interface with the hardware so that we can minimize the effects of implementing the OS design.

The determination of those functions which are to be system support functions usually depends on such considerations as efficiency, convenience, availability, habit, clarity, commonality, and clever programming.

It does seem advantageous, however, to choose system support functions based on criteria which are more standard than those used in conventional systems. That is, if methods were used that were not ad hoc, the properties of a system, both individually and relative to each other, could be more easily understood throughout a given development process.

A proliferation of real-time tactical operating systems has been observed. This observation has lead to the suggestion of a family of operating systems by the Navy [2], and an operating system nucleus by the Army [3].

Attempts at standardizing layers [4] of a system have been made. For example, some HOLs do not permit the combination of HOL code with assembly language code [5]. Even for the Apollo flight software, applications were not allowed to perform OS functions, interpretive code functions [6], man/machine functions [7], or error recovery functions [8] without going through the standard support functions [9]. For such reasons, many designers have advocated hierarchical operating system concepts [10], [11].

A Higher Order Software (HOS) system is specified by hierarchically grouping abstract control structures. Since each control structure is consistent with the properties of HOS, any system constructed from groupings of these control structures, not only is able to maintain interface correctness among hierarchical components, but, system characteristics can be derived so that verification of such a system is a more reliable process. An example of system characteristics for a real-time, multiprogrammed, dynamic scheduling algorithm based on HOS principles has already been shown [12].

In this report, we are concentrating on the subject of operating systems (OS) in terms of the methodology of HOS. This section discusses the basic concepts of resource allocation, to, by and from an operating system, and the techniques of describing these concepts in terms of the specification language AXES. The second section discusses the concept of secure systems. The third section discusses some general data-type definitions for operating systems, two of which (time and address) are used in the fourth section. The fourth section provides example specifications of an operating system machine which uses as a model the Apollo on-board flight software system. The fifth section discusses a concept for a Higher Order Machine (HOM).

The intent of this project is several-fold. It is hoped, that from this effort, people will have a better understanding about the issues of systems design, and, in particular, about the issues of resource allocation. We show here that HOS can be used to define operating systems and that these systems are, by their very nature, secure. Not only can a machine, such as an operating system be secure, but the user systems of that operating system can be secure as well, thus making a complete system definition a secure one. The examples selected from the Apollo operating system are intended to demonstrate the techniques of HOS. The techniques shown here can be used to define other operating systems as well as systems in general. Our next step in the specification of operating systems is to take advantage of the many lessons we learned in the specification of the AGC operating system with respect to machines in general in order to define a complete specification of an ideal operating system as a machine. This

will include the definition of more specific control structures and data types for a family of operating systems. We have used more general HOS control structures and data types for this prototype effort. But, it is clear that certain of the operations defined here for the Apollo system could be used as specification macros for operating systems in general. These macros could then be selected from or added to at will from a library of OS macros. The HOM is envisioned to ultimately replace intermediate machine layers (such as operating systems) and talk directly to HOS systems. The intent of the HOM is to adjust to a system design instead of the system design adjusting to a machine.

2.0 BACKGROUND

The choice of an operating system as a demonstration of HOS is quite appropriate since it has characteristics we wish to demonstrate in describing the concept of an HOS machine. Our first task on this project was to search for a representative operating system to use as a demonstration. In this process, it occurred to us that the operating system we had grown up with (i.e., the Apollo Guidance Computer (AGC) operating system) was a more appropriate candidate to work with than others we had looked at. The reasons for this decision are many: the AGC operating system had flown successfully on several real missions to the moon and back and thus had a few "battle" experiences of its own; we are familiar with the design of the AGC operating system, both from many years of experience as designers and as users of the system; the AGC operating system includes many interesting operating system capabilities from the point of view of specification; the AGC system was intended to be a secure system; the AGC operating system appears to be more efficient than others we have looked at (it was forced to be efficient in order that all the mission programs had enough time and memory to perform their functions). In the context of the functions it performs, the AGC operating system (or its derivatives, such as the HAL run-time package [5] appears to be a simpler and more elegant design (from both an algorithmic and an organizational point of view) than others we have looked at and it is thus easier to work with when comparing the specification of such a system with its implementation; we know well the benefits and the shortcomings of the AGC system.

We will not attempt to describe here the entire AGC operating system, but rather we will show typical algorithms and data types which exist in the AGC operating system machine. The emphasis, here, will not be how to define the AGC operating system, per se, but, rather, we wish to show a technique using portions of this system in order that we can use such an example as an aid in describing other systems (including other operating systems).

This exercise has been an extremely interesting one from several standpoints. Although we were once intimately involved in the AGC system, it took a great deal of time and patience to revisit this environment. This

was mostly due to the fact that the AGC system was poorly documented (although the documentation of the Verification and Validation (V&V) contractor turned out to be better than our own [15]). Our only solution, however, for completely understanding the system (which included, by the way, our own designs and our own coding) was to go back and pour over the original code, which was very clever and difficult to understand. Fortunately, the basic concepts upon which the code was based (which were powerful ones from the standpoint of OS capability) were quite simple, and that fact made it easier to reconstruct the pieces.

The complete AGC operating system contains many capabilities. These include scheduling of processes, error detection and recovery, I/O handling (including uplink to the spacecraft and downlink to the ground mission control), and multilevel display interfaces between the AGC and the astronaut and between the AGC and the ground. To handle its process load, the AGC operating system had several priority systems. In the software part of the AGC we scheduled processes dependent on hardware interrupts, software job priorities, and software task times. We will discuss here specifications which have to do with the scheduling of jobs based on priority (Executive system) and scheduling of tasks based on time (Waitlist system). Both of these machines (i.e. the Waitlist and Executive machines) handled all the types of secure data that are necessary for asynchronous machines.

3.0 RESOURCE ALLOCATION

One of the most difficult problems facing systems designers is that of preparing a design to execute on a particular machine. This process of preparing systems to communicate with each other is called resource allocation. More subtle problems appear when we attempt to redesign our system to fit a machine. Once having done so, we can no longer understand our original design, since the resulting complexity hides the original intent of the design with camouflages of implementation. We not only cannot trace input and output throughout our system design, but we are no longer even sure which input and output is relevant to our original problem. To try to change such a system with a new system requirement is a presumptous notion.

An advantage of an HOS system is that all input and output can be traced throughout a given system definition. A second advantage of HOS is that we can separate the data flow with respect to different layers of implementation. Such a feature allows us to look at only those system interfaces which are relevant at the time when they are relevant, and only at the time they are relevant, in each step of a system design process. Thus we are not forced to redesign one layer when we wish to implement that layer on a machine or when we wish to have it reside dynamically with another process in the same environment. We often hear the complaint from system designers about the problem of attempting to design a system "top-down" and then being forced to add "extra stuff" on a lower level of the top-down design when resource allocation is addressed. This forces an iteration of the design process in order to incorporate that extra stuff back at the top and down through the top-down design to maintain consistency of the overall design. (Some designers merely add that extra stuff without worrying about it because they don't know what to do with it, except to say it came from some other system; but then they lose track of its influence on their own system and the influence of their own system on other systems.)

In order to resource allocate a given system to a particular machine, it is necessary to define both the system and the machine in such a way that a change to one won't affect the other. It is also necessary to

define the machine in such a way that the users of the same machine only affect each other when they are explicitly defined to affect each other.

4.0 SYSTEM LAYERS

With the formalism of HOS, we define a standard set of objects and characteristics of objects that should be described with each system as well as a standard set of nomenclature that should be used in describing these objects. We emphasize in our notation the clear separation of the layers within one system or between systems. In particular, we take great care to distinguish between an instance of a layer (represents one performance pass of a system)* and a layer (represents all performance passes of a system). We distinguish between communication within one layer, which always represents the same instance, and communication between layers, which takes place when an instance of one layer communicates with an instance of another layer (e.g., real-time asynchronous processes). We emphasize the importance of separating the layers of a system development. For example, we distinguish between (1) the system and the definition of that system, (2) the system and the description of that system, (3) the system and the implementation of that system, and (4) the system and the execution of that system.

A machine is a system which executes another system. There are dedicated machines, synchronous machines, and asynchronous machines. A dedicated machine always performs the same function. Thus the "mapping" of an AXES FUNCTION could be viewed as a dedicated machine. A synchronous machine must only execute one system to completion before another system uses that machine. An AXES OPERATION could be viewed as a synchronous machine. An asynchronous machine may execute instances of more than one system before either system reaches execution completion. Thus, an AXES STRUCTURE can be viewed as an asynchronous machine (see Section 8.0 of this report for a discussion of AXES control structure definitions).

The environment of a machine must be secure in that (1) a user should not have to be concerned with any of the details that have to do with its execution,

*as opposed to a level which is a step of refinement (or more explicit definition) within a given instance of a layer.

and (2) a user should not be allowed to have visibility into another user's environment.

In an asynchronous machine there are several types of data, all of which must be maintained as secure data throughout all instances of the machine. These types of data include (1) temporary values which exist for one or more users; (2) values from another machine with respect to the machine itself as a user; (3) values with respect to the variables of the machine itself; (4) values which are functionally related to a previous instance of the machine for a given instance of the machine system; (5) values which are functionally related to a previous instance of the machine for a given instance of a user and; (6) values which are functionally related to a previous instance of the machine for a given instance of another machine.

The definition (dynamic state) of a system is equivalent to the formal semantics of a system. The description (static state) of a system is equivalent to the syntax of a system. The implementation (static state which includes a system, a machine to run that system, and the mechanisms necessary to relate that system to the machine) of a system is equivalent to that same system ready to be exercised. The execution (dynamic state which includes a system, a machine running that system, and mechanisms which relate that system to the machine) of a system is that system being exercised by a machine.

Not only must we be aware of the types of system layers, but we also must be aware of how many different definitions, descriptions, implementations, and executions are possible or potentially possible for one system. Most important, we must determine those states which are necessary and those which are not only unnecessary but which are causing serious difficulties in the development of a system. In order for us to make such wise distinctions followed by wise judgements, we must have available a means for determining both the types and the nature and number of states within each type.

Consider the process of determining the successive layers within a layer type as they relate to the evolving process of a system. Each of these processes is much like the process of a writer relating his thoughts for

a reader to understand. Not only must he convey the information he wants to convey, but he must also convey it in the framework of the reader [14]. Each successive layer is a level of refinement with respect to the previous layer in that such a layer is more dependent on a particular machine.

The first system definition (Figure 1) is a cloud* (or a very fuzzy idea of a system). Each successive definition should be more explicit than the previous definition, in that it is more dependent on a particular definition machine. The process continues until we think we have completed the most primitive definition of the system. Today, processes of refinement (i.e., decomposition) are manual processes.

The first system description (Figure 2) is usually a verbal cloud followed by statements written in English, some sort of "pidgin" English, some sort of so-called "specification" language, a higher order language (HOL), an assembly language, and finally the primitive object code for some machine. Each successive description is more dependent on a particular description machine. In the description process, the support layers communicate with a system in its static state.

The first system implementation (Figure 3) is a cloud (i.e., output and input), the algorithm "machine" that will produce the output from the input, and the operating system(s) machine(s) which operate(s) the algorithms. Each successive implementation is more dependent on a particular implementation machine. This process continues until we reach the primitive machine instructions which are prepared to "run" the results of the most recent implementation.

Finally, when we put together a system, we are ready to exercise that system. In a typical multiprogrammed or multiprocessed system, for example, many different execution configurations of that system are possible--in fact, so many that we can't count them. In the execution process we are concerned with systems communicating with each other dynamically; that is, the objects of one system communicating with the objects of another system.

---

*Bob Fitzwater has suggested the term, "cloud" to describe the initial stage of system development.

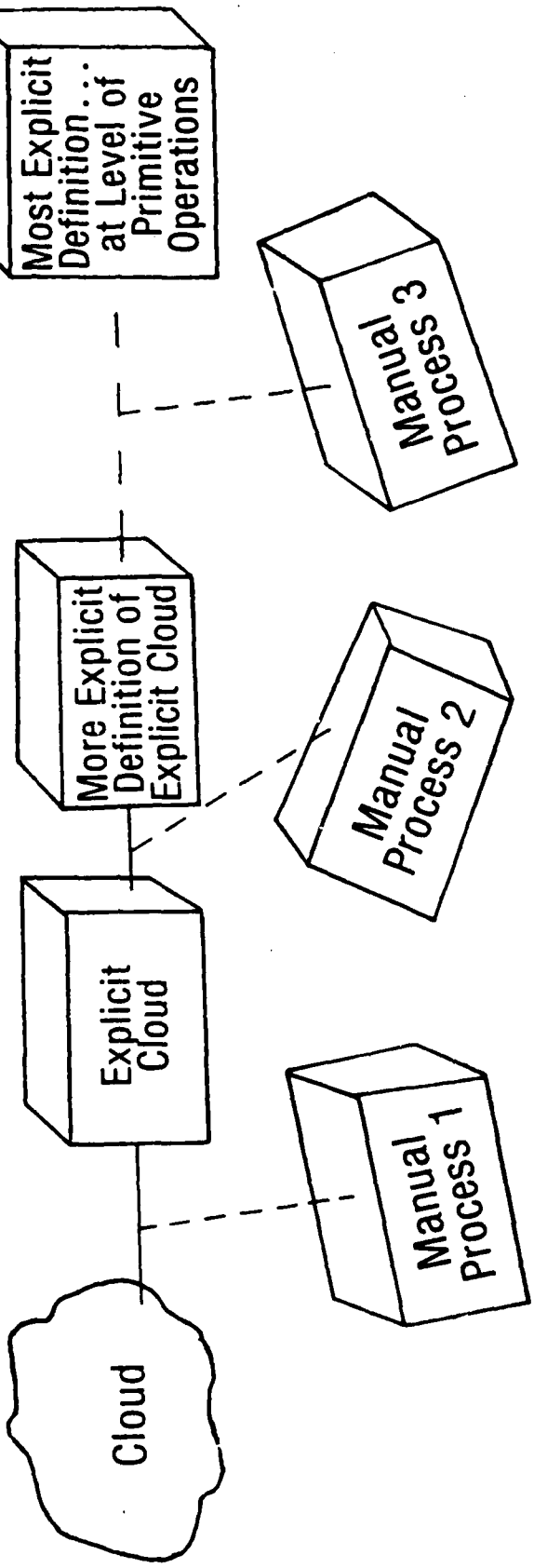


Figure 1. Definition Layers with Respect to Definition Machines
(Each Machine has its own Hidden Data)

HIGHER ORDER SOFTWARE, INC. • 843 MASSACHUSETTS AVENUE • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 661-8900

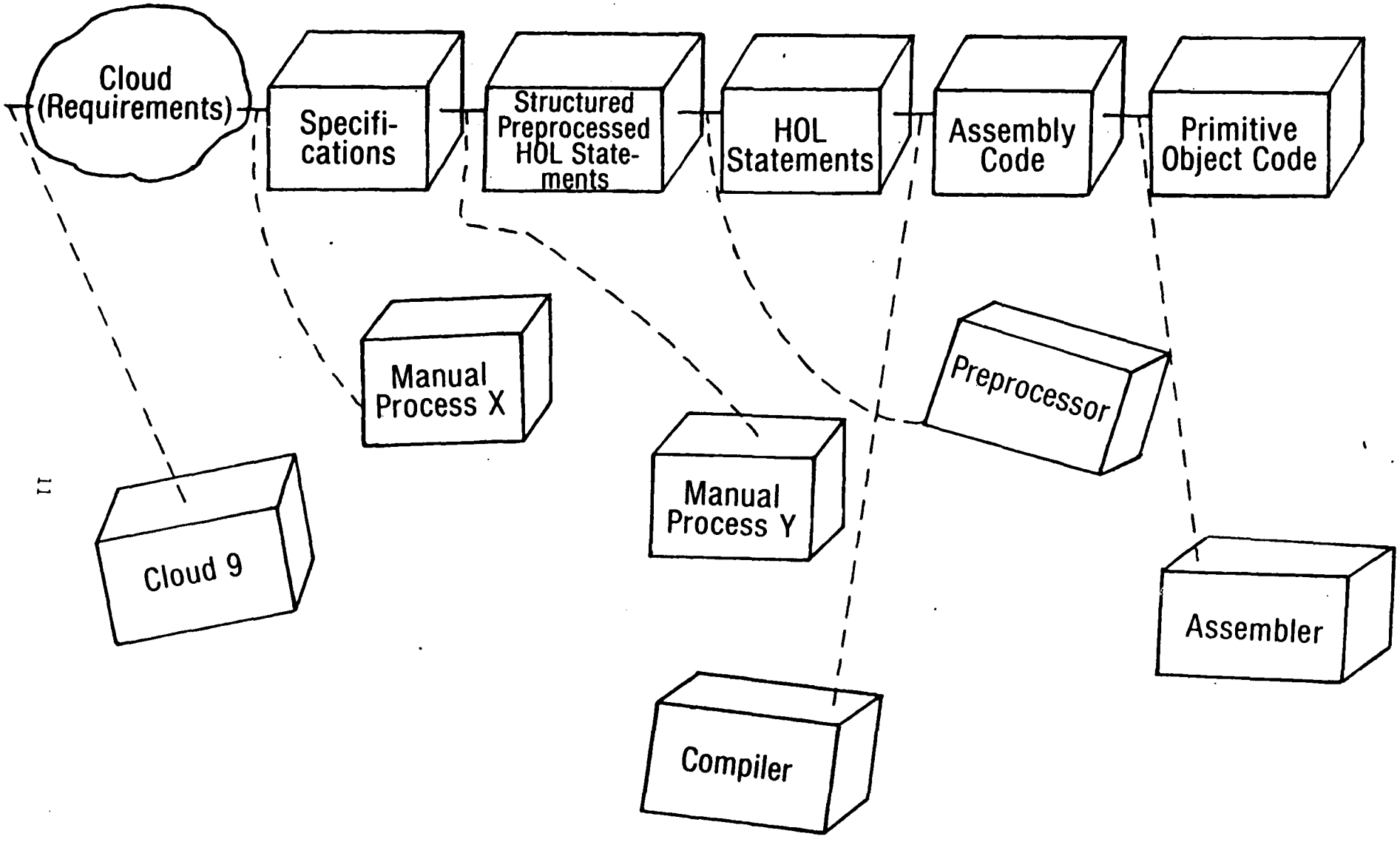


Figure 2, Description Layers with Respect to Description Machines
(Each Machine has its Own Hidden Data)

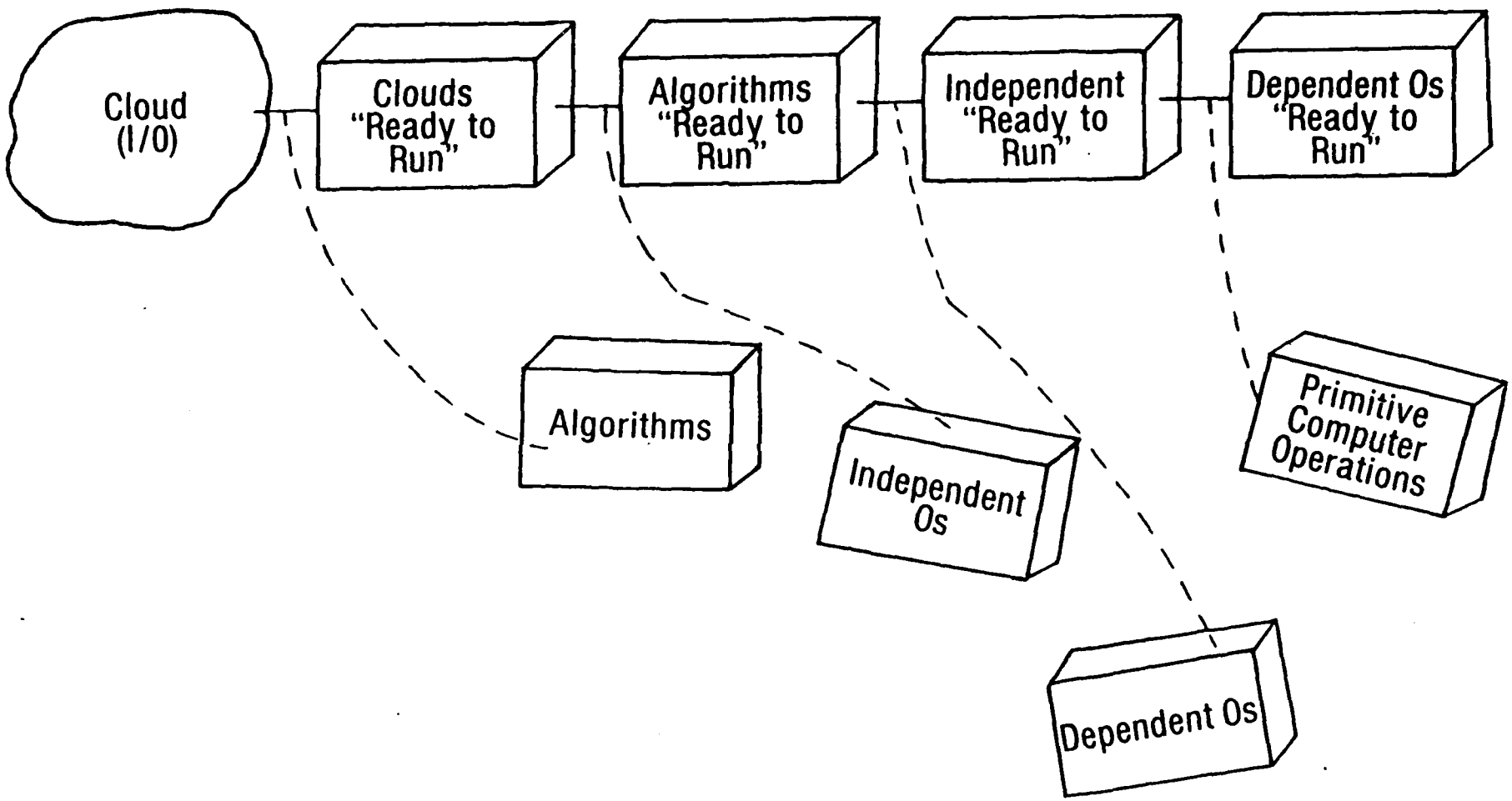


Figure 3. Implementation Layers with Respect to Implementation Machines
(Each Machine has its own Hidden Data)

Sometimes the need for the definition of the layers of a system depends on how the system is to be developed and executed. One project might wish to compile source code before a target system is ready to be executed. Another might wish to interpret the code in real time. Thus, not only must the layers of a system be determined, but the manager must also determine when, how, and where their transformations take place.

No wonder we are all confused! Not only are we not aware that such processes exist (or states within each process), but we also do not know how to clearly separate these processes. A case in point is the manner in which most designers conceptually view the development process of a system (Figure 4). Not only do we start with a cloud, but more often than not, some part of the cloud remains throughout the development process. In fact, sometimes the cloud becomes increasingly larger as we proceed to the final deliverable. Once, however, we know how to make layer distinctions, we can take advantage of such a method to make visible the development of all the components of a system.

Once we are able to define layers explicitly, we will be able to take advantage of more simple concepts for layer communication as opposed to ad hoc methods we apply today. With HOS, we think of layer communication as being one of resource allocation (or assignment). That is, one layer, with respect to another layer, assigns a name to a value or a value to a name. Resource allocation also includes the ability to replace a name by an equivalent name or a value by an equivalent value. Sometimes one layer is produced from another layer by a third layer. Sometimes the description of a layer, as opposed to the layer itself, becomes the object of communication.

The general concepts of layer communication can be related to familiar examples. When two asynchronous processes are in the execution mode, a value from a given process is assigned to a name (or variable) associated with another process. Conversely, that other process assigns a value to the name associated with the first process. When an integer is implemented, a specific representation (or value) for a specific machine is assigned to the name representing the integer. When a compiler compiles an HOL program, it assigns names (or registers) to values in order

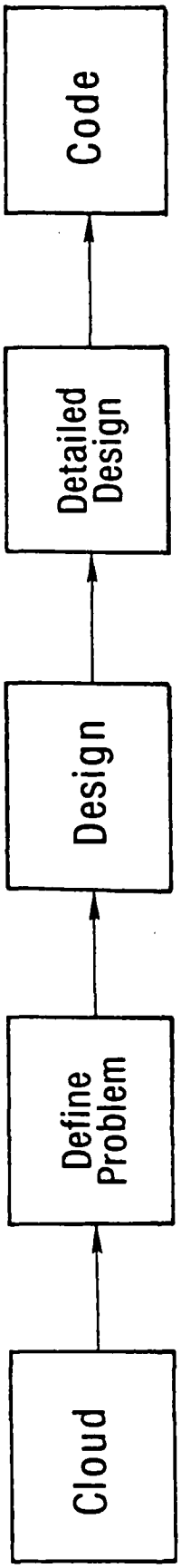


Figure 4. One Viewpoint of a System Development Process (Today)

to translate from one description to another. It also fulfills an implementation function in that operations and data are replaced with specific machine-dependent values. A compiler, in order to operate on an HOL program, must be able to read the description layer of the program as input for the translation process. In addition, it requires further input of its own in order to provide the implementation layer of the program. An OS system has a more complex job in that it is usually required to communicate with both the description of a system and the system itself. When a function is refined to a lower level of more detailed functions, the control integrity of the values and names from the parent layer must be maintained at the layer of the offspring. The layer relationships are defined in HOS with the use of universal operations, or, alternatively, with the WHERE statement in AXES. Several observations can be made from analyzing Figure 1-3: (1) some layers may be unnecessary (for example, why do we need so many description layers in order to talk to the primitive machine); (2) if we truly could make self-contained machine layers, we could transfer a system from one machine to another machine at the same level of detail. (Thus, for example, an integer data type could be moved from one machine to another if it were expressed simply as an integer, I. However, once it has been further specified to be on a particular machine, we then would describe such a data type in terms of its next machine level, i.e., $I_{Machine\_X}$. At this point the integer can only be moved from a machine to another machine with the same architecture as $I_{Machine\_X}$.) (3) We could develop individual machines independently; (4) we should be able to define levels more abstractly in order to hurry the process of getting to "the bottom" of a system; (5) we can tell from all of these various states that there is a natural breakdown for management milestones, whereas today, many of our milestones are not only not too well defined--they are not defined at all.

If we compare our methods of developing systems today with ideal methods, the contrast is overwhelming. Let us not despair, however, for a look into the future will help us to head in the proper direction. Ideally, we want to be able to define, as abstractly as possible, a system first, describe that system in a syntax we can all relate to, verify that description, implement that system for a machine that will talk directly to

HIGHER ORDER SOFTWARE, INC. • 843 MASSACHUSETTS AVENUE • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 661-8900

that system, and collect the machine mechanisms, and only those mechanisms, that are necessary to execute a given system (Figure 5). In such a way we can eliminate dependencies on a particular primitive machine until the very end of a development process. For when we go from one definition to another, from one description to another, or from one implementation to another, we are really resource allocating to only the next machine level. Thus, there is no need to resource allocate for more than one given level at a time (Figure 6).

This allows us the greatest freedom in (1) developing modules independently; (2) transferring modules from one "machine" to another "machine"; (3) making changes to requirements in one module without changing another module; (4) eliminating manual processes (i.e., each manual machine in Figures 1-3 represents a possible process to automate); and (5) eliminating redundant or unnecessary steps. Most important, we have the freedom to change our minds!

HIGHER ORDER SOFTWARE, INC. • 843 MASSACHUSETTS AVENUE • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 661-8900

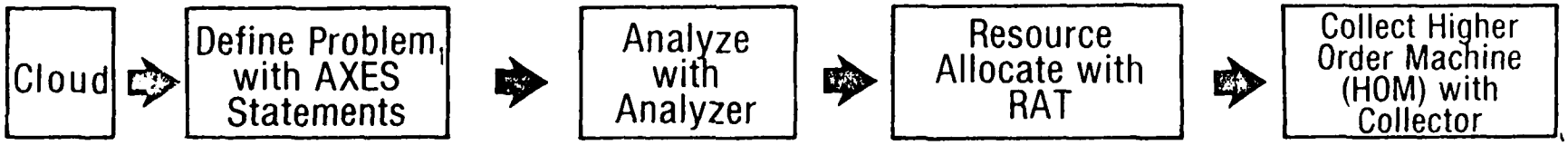
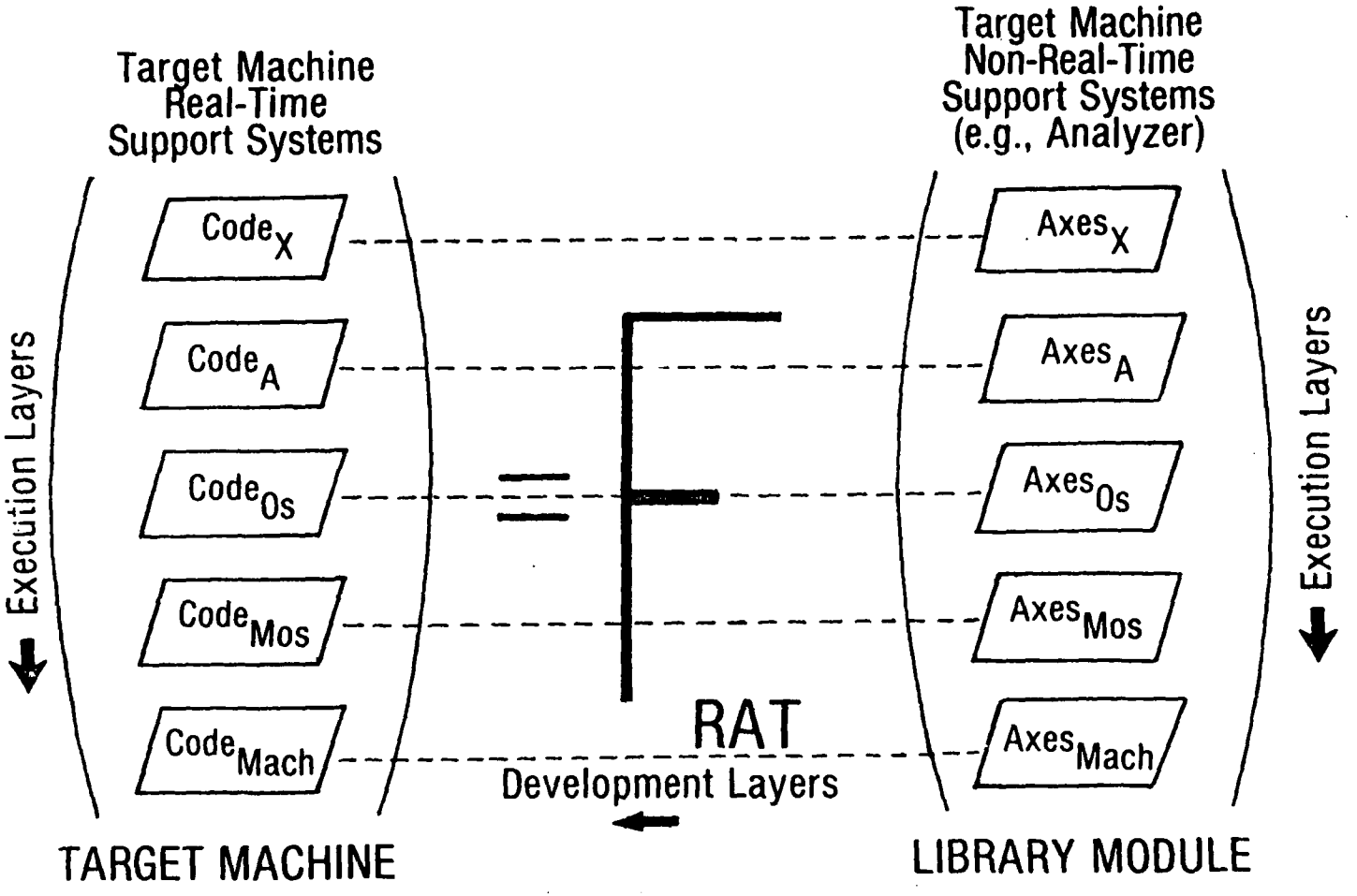


Figure 5. HOS Approach to Software Systems Development

HIGHER ORDER SOFTWARE, INC. • 843 MASSACHUSETTS AVENUE • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 661-8900



KEY:
X = I/O
A = Target System Algorithms
Os = Machine-Independent Os
Mos = Machine-Dependent Os
Mach = Primitive Machine

Figure 6. Development and Execution of a System

5.0 THE IMPLEMENTATION OF A SYSTEM

Looking back on Apollo, hindsight shows us that we had candidates (conceptually, that is) for several different machines, although we did not view them formally as layers at that time and we did not always correctly separate them as layers. Some machines were lower than others in that they were closer to the primitive machine which was the AGC itself (although the internal logic of the AGC could be viewed as still a lower layer in the system). Guidance, for example, was a higher machine layer than the operating system which scheduled it, but the operating system was a higher machine layer than the AGC which scheduled it. Other machines (two asynchronous processes communicating with each other) were on the same level; that is, an instance of the layer of one machine communicated with an instance of the layer of another machine on the same level as an instance of a third layer. In such a case, we say that two or more systems intersect at a third system. Each intersection is an instance with respect to a machine structure which performs their communication where that machine structure is one of the communicating systems. Thus, each system takes turns being a machine for the other.

In the AGC, guidance, navigation, and vehicle control were examples of machine layers which could intersect with each other on another layer. Other less obvious communications (e.g., I/O processing and AGC self check) communicated only in the sense that there was an ordering relationship. In this sense, such processes from the standpoint of a higher layer are independent, but from a lower layer are ultimately dependent, since one is always more important than the other. Together these layers executed in an asynchronous environment in that a higher priority layer could interrupt a lower priority layer and they could execute with their own defined major and minor cycles. In addition, there were several other layers (this included up to eight systems scheduled based on software priorities up to nine systems scheduled based on software times, and up to eleven systems scheduled based on hardware priorities, all of which had possible asynchronous relationships for a given instance of a complete AGC system layer).

When we define a system that is to be executed, that system definition is not complete until we define its state, the change of its state, and the machine that will execute its change of state.

Whenever we want to separate a system from its execution, we create one layer for the system and another layer for the machine that runs that system. Likewise, that same machine can be looked upon as a system with respect to the machine which executes it. This process continues until we arrive at the layers of the primitive machine. Consider potential resource allocation steps for System S where

STEP 1 (determine system): i.e.,

S is System

STEP 2 (determine I/O): i.e.,

S

WHERE S ON (x_1,x_2)*

STEP 3 (determine algorithms): i.e.,

(x_1,x_2)

WHERE (x_1,x_2) ON F

STEP 4 (determine OS): i.e.,

$x_2 = F(x_1)$

WHERE F ON Executive

STEP 5 (determine computer): i.e.,

$Estate_n$ = Executive ($Estate_1$)

WHERE Executive ON AGC

STEP 6 (computer): i.e.,

$Cstate_n = AGC(Cstate_1)$

*WHERE A on B is an AXES statement that means A executes on Machine B and the execution of A represents an instance of Machine B.

Here we have six different layers for System S. If we want to move F to another operating system, we change Step 4 to name a new OS for F and replace Steps 5 and 6. If we wish to design a new algorithm for the same computer, we change Step 3 and replace F with a new algorithm and then replace Steps 4, 5, and 6. If we want to move to a new computer and our operating system is independent of its computer (as it ideally would be), we can change Step 5 to refer to another computer and we then can replace Step 6.

6.0 THE SPECIFICATION OF AN IMPLEMENTATION OF A SYSTEM WITH HOS

In order to implement a system, it is desirable to adhere to the following requirements: the users should not be aware of each other's secure data or each other's timing; a change to one user should not unintentionally affect the other; a change to a user should not affect the machine that executes that user; a change to the machine should not affect the users which will execute on that machine, and the user should not be aware of the machine's secure data or timing.

A system defined in HOS has all the necessary information for a machine to use in executing that system. Consider System R (Figure 7) as an example of a system which has been defined to execute on machine OS. If we view the HOS machine, OS, with respect to System R from the point of view of the order in which that system is to be executed, the relevant information of System R to the OS would be that which is shown in Fig. 8.

In System R, R is specified to invoke A and B. It is clear from the control map that the process B must precede A since A needs B's output to execute. Similarly, A controls C to precede D, since D needs C's output. Thus the OS machine would assign a higher priority to B than A and a higher priority to C than D. R, as a controller, is assigned a higher priority than B; and A, as a controller, is assigned a higher priority than C.

Although the offspring of C do not depend on each other for inputs, C controls G to precede F which precedes E. In addition, E is controlled to start in ΔT after F*. Here G is assigned a higher priority than F and F a higher priority than E in that they must execute in this order if they are in the same processor, although if C were in a multiprocessor, functions E, F, and G could run concurrently as long as there were sufficient processors available. F controls I or L to be scheduled in ΔT_1 or ΔT_2 with respect to the time of F's invocation. In this case, only one of the functions would be executed by OS.

---

*$E_{\Delta T}$ would be defined by an AXES STRUCTURE definition.

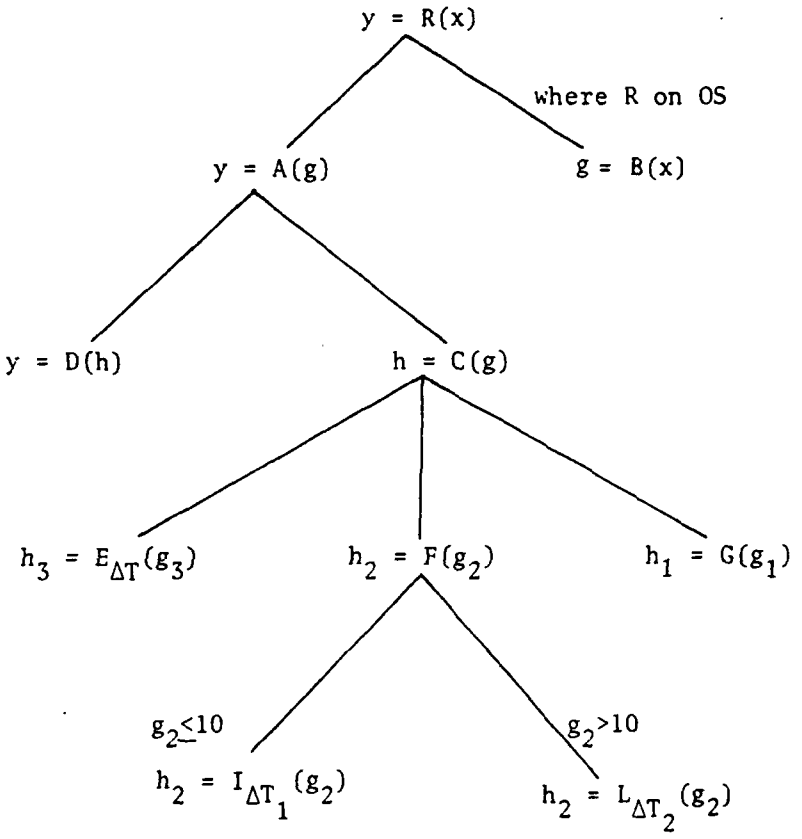


Figure 7. Example of an HOS System Allocated to Execute on an HOS Machine

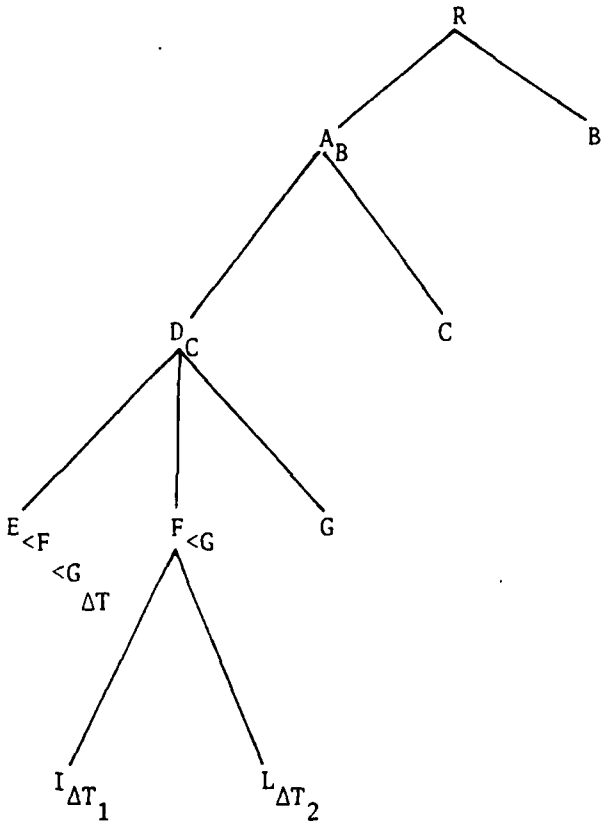


Figure 8. System R as Viewed by Machine OS

In the above example, if one wanted to change to another operating system, we could do so by simply changing the WHERE statement (for example, WHERE R ON OS_1 or WHERE R ON HOM). A change made to any part of System R does not affect System OS. Conversely, a change to System OS does not affect System R. The scheduling of the functions in System R are completely managed by the OS machine. Thus, ordering considerations as well as data with respect to execution are completely invisible to the user. In essence the control map of the userserves as all the necessary information to the machine to carry out its execution (i.e., the order in which the functions are to be performed can be determined by the machine receiving as input the nodal families of a system (c.f. Appendix). The OS machine is an AXES STRUCTURE where each instance of that STRUCTURE is analogous to an instance of execution on the OS machine (Figure 9). In this figure the universal operation K_{FNAME} defines a particular constant operation where a value for "FNAME" is supplied by the user (c.f. Section 8.0 of this report).

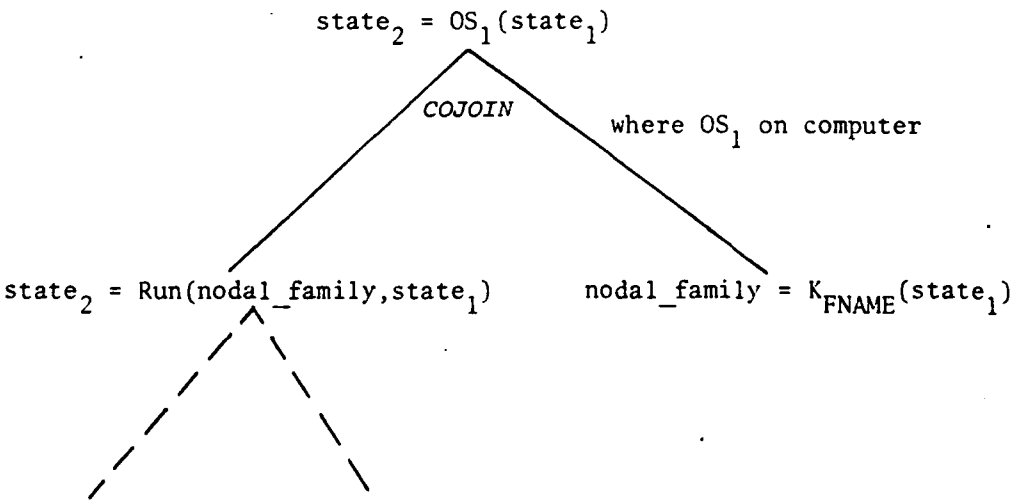


Figure 9. An Instance of an OS Structure Definition

This example is oversimplified since it does not show the recursive nature of this machine (c.f. Section 8.0 in this report). In addition, in a real machine we would need a mechanism to turn the machine on or off, provide error detection and recovery, snapshot and rollback, and redundancy management.

7.0 ANALYSIS OF THE AGC OPERATING SYSTEM

In the AGC operating system environment, there were several different priority systems (machines) used by the users and managed by the users with respect to interfaces between each priority system and within each priority system.

The Executive schedules processes based on priority. A higher priority means that if two processes are scheduled at the same time, the higher priority process is given precedence over a lower priority process. The Waitlist schedules processes based on time. Thus a process scheduled for an earlier time would take precedence over one scheduled to take place at a later time. In the AGC, a Waitlist process scheduled for immediate processing takes precedence over an Executive process scheduled for immediate processing, i.e., Waitlist processes could interrupt* Executive processes, although there was an exception when the Executive interrupt structure inhibited other interrupt structures**. The AGC allowed an Executive scheduled process to inhibit interrupts and then to release such an inhibit. The instructions which could execute between an INHINT and RELINT command in the AGC are analogous to the "critical sections" concept mentioned in the work of Dijkstra [10]. However, such an operating system, the earlier operating system (i.e., the AGC), had a mechanism which prevented deadlock (i.e., the INHINT mechanism and RELINT mechanisms were hidden from the user).

The AGC Executive uses the starting address of a process and its priority as input. The AGC Waitlist uses the starting address of a process and its "time to go" as input. In order to show a complete specification of the AGC operating system as a machine, we would need to show all components

---

* In the AGC an interrupt was either an exchange of a temporary program counter and an active program counter or an exchange of a program counter from one priority structure and a program counter from another priority structure, i.e., (PC_2, PC_1) where "PC_1" and "PC_2" represent values. (See XCH operation described in Section 8.0 of this report.)

** The Executive could inhibit all interrupts except that interrupt which overrode all other interrupt systems. This interrupt was used for emergency situations.

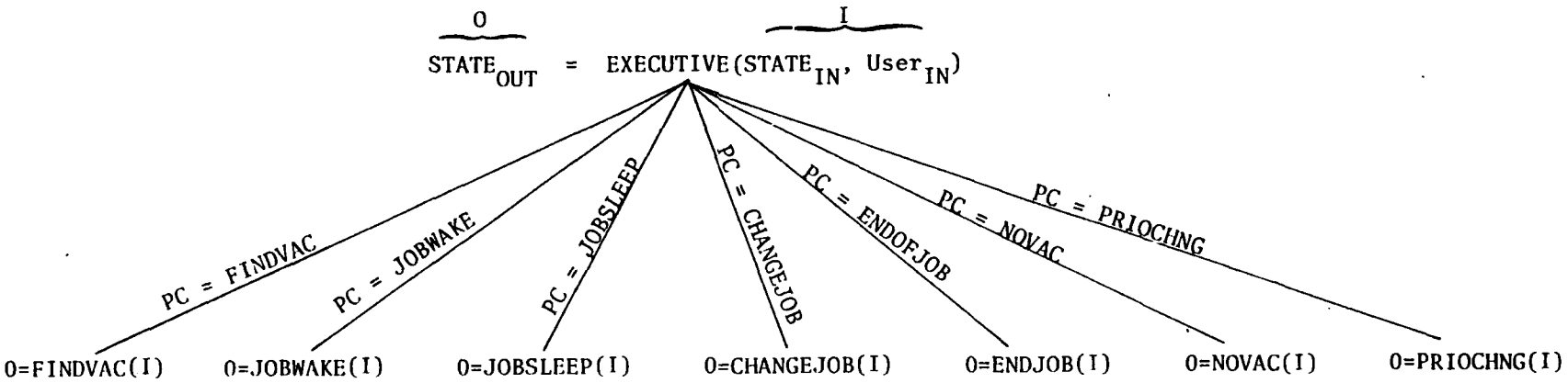
of the AGC operating system and their relationships to each other. The AGC operating system, likewise, could be viewed as having machines of its own. For example, the Waitlist is a machine and the Executive is machine, and these machines both talk as users to the AGC itself. We have not attempted to show here all of these interrelationships, although it would be a very interesting exercise to do so.

In Figure 10 we show a description of an Executive system as a machine similar to the top-level Apollo Executive system. The Executive had several different types of scheduling functions. These included Findvac which scheduled jobs with larger memory requirements and Novac which scheduled smaller size jobs; Spvac was used to schedule larger jobs with a special size address; Endofjob was used to terminate a job; Priochng was used to change the priority of an already existing job. Jobsleep was used to change an active job in the queue into an inactive job in the queue. Jobwake was used to activate a sleeping job. Changejob was used to check for a higher priority job and thus served as a detector of possible interrupts.

The problem with this Executive was that the users, themselves, decided which Executive function they needed, instead of the Executive machine. In addition, the user, instead of the Executive machine, decided which priority to use. In both cases, an Executive machine would be much more qualified to determine these parameters since it has all of the information of the other users available that the user does not have.

The Waitlist system (Figure 11) had more subtle problems for the user to worry about. For not only did the user have to worry about coordinating his own timing with respect to other users, but he also had to be aware of the behavior of the Waitlist with respect to its own machines and with respect to the interfaces between these machines.

Not only did users of Waitlist have to coordinate themselves with each other and users of the Executive had to coordinate themselves with each other, but these two sets of users had to also be coordinated by the users themselves. Thus the users had to coordinate other users of the scheduling mechanisms as well as be aware of how these various mechanisms interfaced with each other. They, therefore, had to coordinate themselves with respect to both synchronous and asynchronous timing aspects of their environment.



WHERE PC = PROGRAM_COUNTER;

WHERE $STATE_{OUT}$ = $VACSTACK_{OUT}$, $QUEUE_{OUT}$, $PROGRAM\_COUNTER_{OUT}$;

WHERE $STATE_{IN}$ = $VACSTACK_{IN}$, $QUEUE_{IN}$, $PROGRAM\_COUNTER_{IN}$;

WHERE $USER_{IN}$ = PRIORITY, ADDRESS, PROGRAM_COUNTER;

Figure 10. Top-Level Description of an Instance of the AGC Executive Machine as part of a Machine System

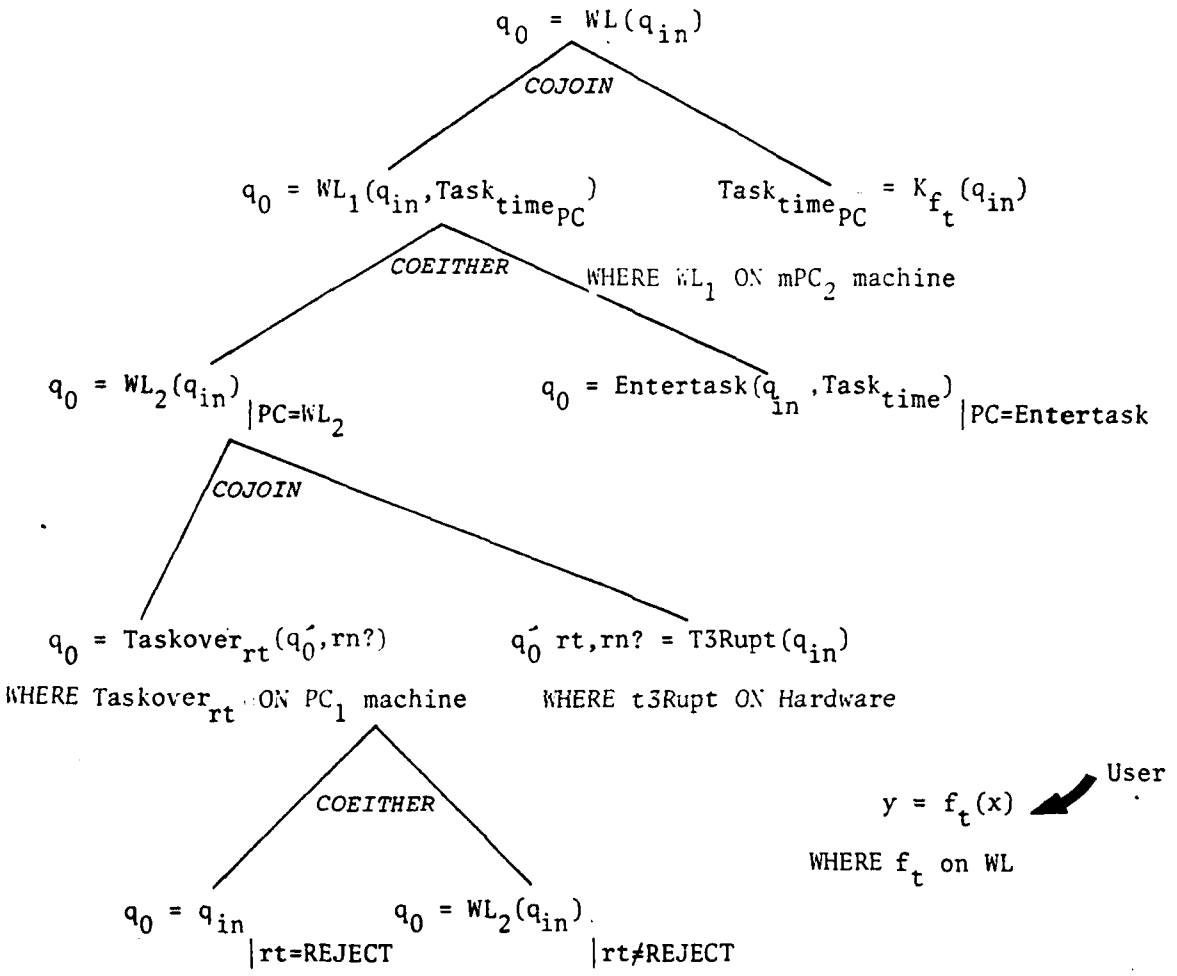


Figure 11. An Instance of the Waitlist Machine with Respect to the Waitlist Machine Environment

Consider the following Apollo scenario:

(1) User A schedules B at priority 10*.

(2) User A schedules C at priority 20.

(3) User A schedules D in 2 seconds.

(4) B schedules itself at priority 12 for the next cycle in 10 seconds.

(5) C schedules D at priority 21 for the next cycle in 1 second.

(6) User A was scheduled by another process at priority 5.

(7) B changes the priority of C to priority 9.

(8) C puts B to sleep.

(9) A wakes B up.

From this scenario, it is first of all very difficult to know without quite a bit of analysis if all of these schedules make sense, but in examining them further, we see there are many latent problems here. And, in fact, many of them happened sometime in the development process of Apollo. Examples** of these potential problems are:

(1) User A loses control due to scheduling B at a higher priority then itself.

(2) B may not finish before another B is processed. Then we have B conflicting with itself.

(3) B has made C be less important but this violates A's original intention.

(4) None of these users are guaranteed that all the priorities under A's control are correctly related.

(5) A different OS function is called to schedule timed processes than the one used to schedule priority processes.

(6) A different OS function is called for scheduling different kinds of priority jobs.

(7) A change to one process priority could undo its originally intended relationship to other processes.

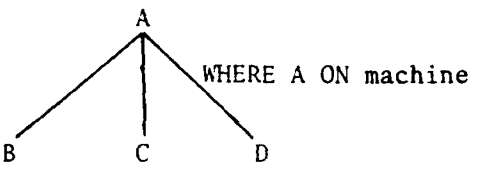
(8) B, C, and D terminate without A knowing it.

(9) A is not aware of other schedules within A's system environment.

*Lower numbers indicate lower priorities.

**An exercise is up to the reader to discover other problems here.

Using HOS design, the following would have happened:



(1) A controls the priorities of B, C, and D with respect to each other.

(2) A always has a higher priority than B, C, and D.

(3) None of these functions can control themselves.

(4) Only A can invoke B, C, and D.

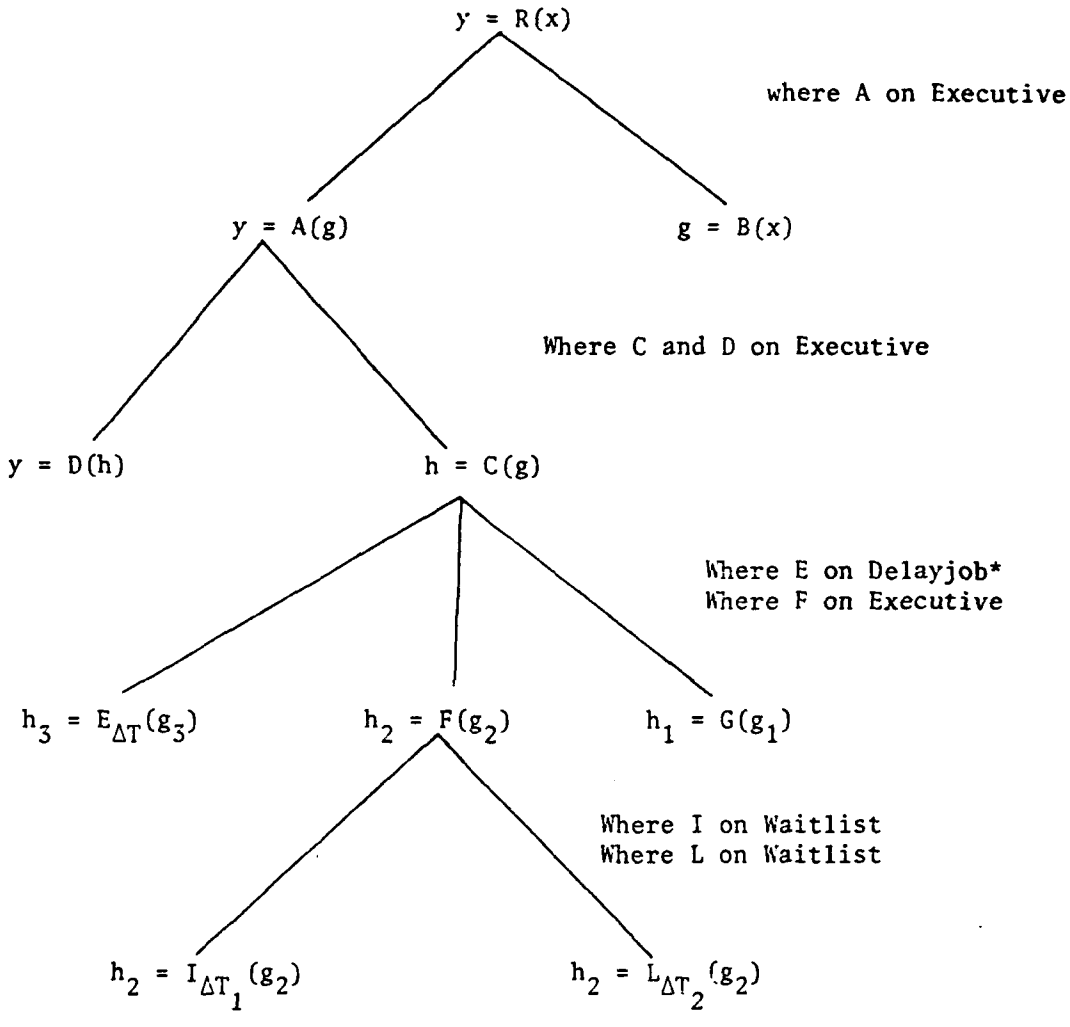
(5) One machine would be used to coordinate the scheduling of all processes in System A.

(6) The machine would maintain relative priorities and thus provide for automatic reconfiguration.

We discovered more explicitly the problems of the AGC operating system by attempting to have an HOS type user such as System R (Figure 7) interface with supposedly more than one machine (i.e., different parts of the AGC operating system) where these different machines should have been coordinated by only one machine with respect to the user. In Figure 12, we show a resource allocation of System R to the AGC machine environment as it existed at the time of Apollo.

In this case, R, A, C, D and F are scheduled on the Executive; I and L are scheduled on the Waitlist; B and G are implicit calls; and E is scheduled by a hybrid (i.e., Delayjob which contains an Executive and Waitlist mechanism to schedule a job in ΔT seconds).

This example does not demonstrate all the potential interface problems since in the AGC the schedule statement also includes scheduling information such as absolute priority. This example shows only the Waitlist and Executive as machines and not any of the other machines which existed in Apollo. But, it is clear that there is no mechanism in this example to coordinate both the timing of the users with respect to each other and the timing of the users with respect to the interfaces between the Executive and Waitlist.



Example I2. Resource Allocation of Machines to one Implementation Layer of AGC Operating System

*Delayjob contains an Executive and a Waitlist mechansim to schedule a job in ΔT seconds

8.0 AXES SPECIFICATION TECHNIQUES

AXES is a formal notation for writing definitions of systems. These systems include systems which are mechanisms for defining other systems. Thus, for example, we could define a set of specification "macros" which collectively could form a language for defining a system or family of systems. Since each language statement would be a definition "macro" based on an integrated HOS control hierarchy, the resource allocation to a particular machine could then be addressed independently from the definition of the system. Although it is not a programming language, AXES is a complete and well-defined language capable of being analyzed by a computer. AXES is intended to provide commonality between systems. Although users will have flexibility to choose different building blocks, these building blocks, when "compiled," will be brought to a common meeting ground with all other users of AXES.

The syntax of AXES [15] provides the mechanisms to specify control structures and data types. The purpose of AXES is to be able to express a system specification which is equivalent to that same specification expressed graphically as an HOS control map [16]. Control structures have three forms in AXES: structures, operations, and functions. Whereas a structure is a relation on a set of mappings, i.e., a set of tuples whose members are sets of ordered pairs, an operation is a set of mappings which stand in a particular relation. An operation results, mathematically, from taking particular mappings as the arguments (nodes) of a structure. By a function, we mean a set of mappings which stand in a particular relation for which particular variables have been chosen to represent their inputs and outputs. Whereas structures and operations can be described as purely mathematical constructs, a function is a hybrid, consisting of a mathematical construct and a linguistic construct, i.e., an assignment of particular names of inputs and outputs. Note that our use of "function" is slightly different from what is meant by "function" in mathematics. For the latter notion we use the term "mapping" throughout this paper.

In AXES, a new data type can be defined simply in terms of the operations that are to be performed on the data [15]. The primitive operations are not defined in terms of other operations, but in terms of each other.

That is, a data type is defined algebraically rather than operationally by making true statements (or axioms) about the equality of two control structures in which all the nodes are operations. Each such control structure is defined in terms of primitive operations of the data type of interest or of previously characterized primitive operations of another data type (previously characterized primitive operations include universal primitive operations that have been defined, each of which is associated with any member of any data type).

The axioms associated with the definition of a data type are only those we need to characterize the data type. There are, of course, other operations that we find useful for other purposes. We are free to define any operation we want on an already-defined type as long as the operation definition is consistent with the axioms of the type. A new operation can be characterized either as an OPERATION or as a DERIVED OPERATION.

In AXES, we specify the behavior of an operation without specifying its decomposition by writing it as a derived operation, i.e., by means of true statements that describe the behavior of the operation with respect to other already-defined operations. Either kind of operation could be written as a control map, if desired. They differ in how they are specified, not in what they are. What distinguishes both of these kinds of operations from primitive operations on their data type is that their existence is provable mathematically from the existence of the primitive operations and the axioms of the type. In fact, if an OPERATION (which defines a function) and a derived operation (which defines the behavior) are both used to define the same function, the behavioral properties can be checked against the refinement properties to prove the correctness of a definition.

In describing AXES we will use variables and constants themselves to make statements about the values they name, and we will use the names of variables and constants to make statements about the variables and constants themselves.

To differentiate an object from its name, we introduced the "use-mention distinction" [17] in AXES [15]. That is, we can talk about an object only by using a name of the object. (To talk about a man, for example, we have to use a sentence that contains the man's name, not the man himself.) The notation conventionally used for this is enclosure within quotation marks. To form the name of a given name (or written symbol of any kind), we include that name (or symbol) in quotation marks. (Successive embedding of quotation marks can be used if we want to talk about names, names of names, and names of names of names.)

In AXES, a constant symbol is the name of a particular value and corresponds to a proper noun like "John." A variable is the name of more than one possible value and corresponds to a common noun like "a man."

For example, in Figure 13, the top-most box is a description of part of AXES itself. The top-most box describes the AXES objects required to define a STRUCTURE in AXES. The sentence

"STRUCTURE:" y "=" S "(" x ")";"

makes a statement about values by using the variable "y", "s", and "x" and about constant symbols by using the quotation-marked symbols such as "'STRUCTURE'", and "'='".

The middle box encloses an AXES object itself; that is, the middle box encloses the definition of a language statement derived from the definition of an AXES module. The Composition (Cn) STRUCTURE, defined in Figure 13, is one of the three HOS primitive control structures. Each primitive control structure has been defined as a STRUCTURE with AXES [15]*. The middle box encloses an instance of the layer that the top-most box

*The syntax for set partition is

$$^{2}y = f_1(^{1}x) \text{ OTHERWISE } ^{2}y = f_2(^{2}x);$$

WHERE PARTITION OF (x,y) IS ANY PARTITION;

Here, the left superscript indicates a member of a member of a partition of "x". The syntax for class partition is

$$y_1 = f_1(x_1) \text{ INCLUDE } y_2 = f_2(x_2);$$

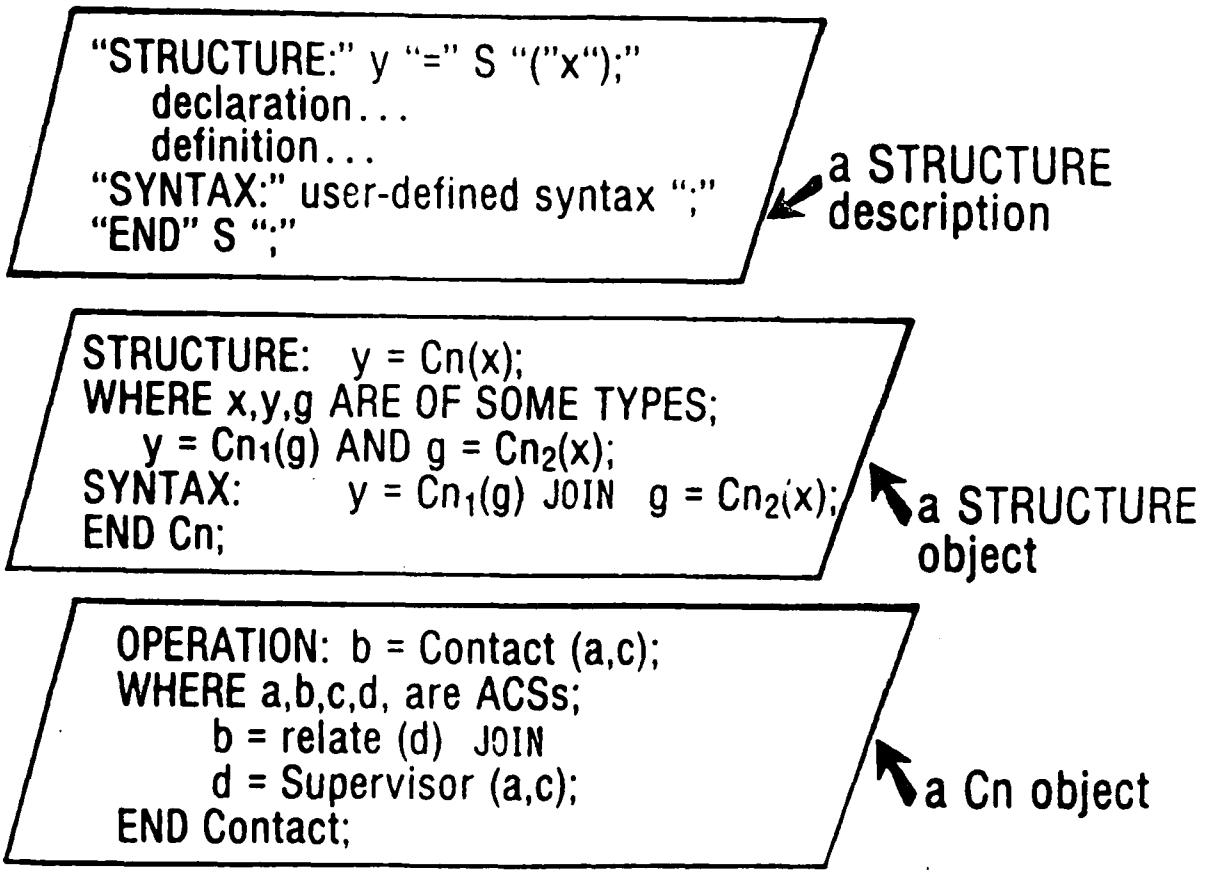


Figure 13

An Example of Abstract-Control-Structure Definition Layers with Respect to AXES

represents. If we could describe all of the structures that could possibly exist, then the complete set of structures would be the layer that the top-most box describes.

When a STRUCTURE in AXES is defined, the designer supplies the syntax (or description) so that a user of that structure can describe particular mappings that stand in the relation. For example, the bottom-most box in Figure 13 encloses an AXES object that is an instance of the Cn structure described in the middle box of Figure 13; that is, the bottom-most box is the definition of a system derived from the definition of a language statement, derived from the definition of an AXES module. In the bottom-most box, "b = Relate(d)" describes a particular function that "y = $Cn_1(g)$" represents in the middle box. Likewise, "d = Supervisor(a,c)" represents an instance of "g = $Cn_2(x)$". In the middle box, the objects that "y", "g" and "x" represent are described in the statement

WHERE x,y,g ARE OF SOME TYPES;

This statement means that x,y, and g are variables whose values are of an unspecified data type. In the bottom-most box the WHERE statement is used to specify a particular data type, and the operation, Contact, is a particular mapping.

Other control structures can be derived from already defined control structures and operations that operate on variables of any type. Operations that operate on variables of any type are called <u>universal operations</u>. Primitive universal operations are defined as

(1) $x = Clone_1(x)$

(2) $(x,x) = Clone_2(x)$

(3) $CON = K_{CON}(x)$

(4) $x_1 = id_1^2(x_1,x_2)$

(5) $x_2 = id_2^2(x_1,x_2)$

(6) $(x_1,x_2) = St(x)$

(7) $x = T(x_1,x_2)$

(1) and (2) are used to specify more than one variable with the same value. (3) is used to choose a constant symbol. (4) and (5) are used to select the value of one of a set of variables. (6) and (7) are related by T(St(x)) = x. These are used to create a value of a data structure from a value of a data type (i.e., St) or to create a value of a data type from a value of data structure (i.e., T).

Universal operations have as their bottom nodes, universal primitive operations. The universal operations

$$y = id_a^b(x)$$

$$y_1, y_2 = XCH(x_1, x_2)$$

$$y = Clone_n(x)$$

are defined here because they are then used to define control structures whose syntax is used to define the operating system functions in Section IV.

Universal operations are defined as STRUCTURES in AXES because they operate on variables rather than values. The first non-primitive universal operation defined here, $y = id_a^b(x)$, is used to select particular variables out of a set of variables.

```
STRUCTURE:  y = I(x)

WHERE c,a,e,e ARE SETS (OF NATURALS);
           1

WHERE b,d,j,p,m,a ,a  ARE NATURALS;
                 2  1

WHERE x,x,x,x,x,x,x,x,x,y,g,z,h ARE OF SOME TYPE;
       1 2 3 4 5 6 7 8

WHERE (a1,a2) REPLACES a;

WHERE  y REPLACES (y1,y2);

WHERE (e1,e2) REPLACES e;

y = I1(x,x,a) JOIN (x,x,a) = I2(x);
      1 4          1 4

(x,x,a) = I1 (x,x) JOIN (x,x) = Clone2(x);
 1 4       2 2 3       2 3
```

$a = I_{1_{1_2}}(x)_2$ INCLUDE $(x_1, x_4) = Clone_2(x)_3$;

$a = I_{1_{1_{1_2}}}(e,b)$ JOIN $(e,b) = K_{(c,d)}(x)_2$;

$a = I_{1_{1_{1_{1_2}}}}(e_1,b_1,b_2)$ JOIN $b_1,b_2 = Clone_2(b)$ INCLUDE $e_1 = Clone_1(e)$;

$a_1 = I_{1_{1_{1_{1_{1_2}}}}}(e_{1_1},b_1)$ INCLUDE $a_2 = I_{2_{1_{1_{1_2}}}}(e_{2_1},b_2)$;

$a_1 = K_{REJECT}{}^{1}(e_{1_1},b_1)$ EITHER $a_1 = id_1^{2}{}^{2}(e_{1_1},b_1)$;

PARTITION OF (e_{1_1},b) IS

$^{1}(e_{1_1},b) | a>b$,

$^{2}(e_{1_1},b) | a \leq b$;

$a_2 = K_{REJECT}{}^{1}(e_{2_1},b)$ INCLUDE $a_2 = I_{1_{1_{1_2}}}{}^{2}(e_{2_1},b)$;

PARTITION OF (e_{2_1},b) IS

$^{1}(e_{2_1},b) | e_{2_1}$ = REJECT,

$^{2}(e_{2_1},b) | e_{2_1} \neq$ REJECT;

$y_1 = I_{1_1}(x_1,a_1)$ INCLUDE $y_2 = I_{2_1}(x_4,a_2)$;

$y_2 = id_2^{2}{}^{1}(x_4,a_2)$ EITHER $y_2 = I_{2_{2_1}}{}^{2}(x_4,a_2)$;

PARTITION OF (x,a_2) IS
4

$^{1}_{4}(x,a_2) \mid a_2 = \text{REJECT},$

$^{2}_{4}(x,a_2) \mid a_2 \neq \text{REJECT};$

$y_2 = I_1(x,x,a_2)$ JOIN $x,x = \text{Clone}_2(x)$ INCLUDE $a_{12} = \text{Clone}_1(a_2);$
7 8 1 7 8 1

$y_1 = I_{1_1} {}^{1}_{1}(x,a_1)$ EITHER $y_1 = I_{2_1} {}^{2}_{1}(x,a_1)$ EITHER $y_1 = K_{\text{REJECT}} {}^{3}_{1}(x,a_1);$

PARTITION OF (x,a_1) IS
1

$^{1}_{1}(x,a_1) \mid a_1 = 1,$

$^{2}_{1}(x,a_1) \mid a_1 \geq 2,$

$^{3}_{1}(x,a_1) \mid a_1 = 0;$

$y_1 = I_{1_1} (g)$ JOIN $g = id_1^2 {}^{1}_{1}(x,a_1);$

$y_1 = id_1^2 (g_1,g_2)$ JOIN $(g_1,g_2) = \text{St}(g);$

$y_1 = I_{1_1}(z,j)$ JOIN $z,j = I_{2_1} {}^{2}_{1}(x,a_1);$

$(z,j) = I_{1_2 2_1} (x,a_1,x_6,a_2)$ JOIN $(x_5,a_1,x_6,a_2) = \text{Clone}_2(x,a_1);$
5 1 1

$j = I_{1_2 2_1} (x,a_1)$ INCLUDE $z = I_{2_1 2_1} (x,a_1);$
5 1 6 2 1

$j = p\text{-}m$ JOIN $m = K_1(x)$ INCLUDE $p = \text{Clone}_1(a_1)$
5 1

JOIN $a_1 = id_2^2 (x,a_1);$
2 5 1

$z = id_2^2 (h_1,h_2)$ JOIN (h_1,h_2) = St(h) JOIN h = $id_1^2 (x,a_1)$;
6 2

SYNTAX: $y = id_c^d(x)$;

END I;

In the use of this structure, a value for (c,d) defines a particular K_{CON} operation which, in turn, is used to define the number of components of "y", each component having the same value as the particular component of "x". For example, an instance of this structure is defined by replacing the value of "c" by a set of integers, the value of "d" by an integer, the value of "x" by a variable or set of variables, and the value of "y" by a variable or set of variables. An instance of this structure might look like $(e,j) = id^5_{(2,4)} (g,h,i,j,k)$. In this instance we have replaced "c" by "(2,4)" and "d" by "5", which defines $(e,b) = K_{((2,4),5)}$ so that "e" has the value "(2,4)" and "b" has the value "5". Here, "y" would have the value "(1,j)" and "x" has the value "(g,h,i,j,k)". Note also that the definition is constrained so that each value of each component of "c" must be less than the value of "d" to assure that the choice of values falls in the range of the number of variables available to choose from. This is accomplished by specifying a particular component of "a" to have the value REJECT if the value of that particular component is greater than the value of "d". REJECT is a value which is a member of every data type in AXES. Its purpose is to be able to specify error conditions and to be able to recover from these errors within the specification of a system.

Although the definition of $y = id_c^d(x)$ is quite complex (because we must use here only primitive universal operations and primitive control structures) once defined it can be used to define other structures. In what follows, the use of this structure is shown to simplify the definition of other structures.

We can now define a structure whose syntax can be used to define more than one system having access to the same value. Here, we use the universal operation

$$y = id_a^b(x)$$

as well as the universal primitive operation

$$(x,x) = \text{Clone}_2(x)$$

to determine the meaning of the relationship among the unspecified functions that appear as bottom nodes of the structure definition.

STRUCTURE: $y = J(x)$;

WHERE y,g,w,h ARE OF SOME TYPE;

WHERE b IS A NATURAL;

WHERE a IS A SET (OF NATURALS);

$y = J_1(g,w)$ JOIN $(g,w) = J_2(x)$;

$(g,w) = J_{1_2}(x,x)$ JOIN $(x,x) = \text{Clone}_2(x)$;

$g = J_{1_{1_2}}(x)$ INCLUDE $w = id^b_a(x)$;

$g = J_{1_{1_2}}(h)$ JOIN $h = id^b_c(x)$;

SYNTAX: $y = J_1(g,w)$ COJOIN $g = J_{1_{1_2}}(h)$;

END J;

In using the syntax of a structure, an instance of the layer of the structure definition can be obtained. In the COJOIN structure, there are actually four unspecified mappings besides the top node: J_1, $J_{1_{1_2}}$, id^b_a, id^b_c. But in the use of the COJOIN, the value of "w" and "x" uniquely determines the particular id^b_a function. Likewise, the value of "h" and "x" uniquely determines the particular id^b_c. Thus, only J_1, $J_{1_{1_2}}$ need appear in the syntax of the COJOIN definition. The collective set of values that replaces the variables described in the syntax can be traced to each node of the structure definition. For example, if

(a,b) = F(r,t,s)

is defined as

(a,b) = A(p,q,r) COJOIN (p,c) = B(r,t)

The first and second statement collectively form an instance of the COJOIN structure, as described in Figure 14. The arrows represent the values of the variables of the COJOIN definition.

In this example, "x" has the value "(r,t,s)"
"J" has the value "A"
"y" has the value "(a,b)"
"g" has the value "(p,q)"
"w" has the value "r"
"$J_{1_{1_2}}$" has the value "B"

and, since the input to F has three components, "b" in the structure definition has the value "3", since "w" has the value "r", which has one component, "a" in the structure definition has the value "1", and so on. The structure syntax names the objects necessary so that an instance of the structure definition is obtained. Any instance of a structure must itself be an HOS system.

In the COJOIN structure, systems that communicate with each other can access the same value. Likewise, we define other structures, one so that independent subfunctions can access the same value (the COINCLUDE), and one so that subfunctions whose invocation depends on the value of the controller's input set need not access the entire set of variables of the input set (the COEITHER).

STRUCTURE: y_1, y_2 = COIN(x);

WHERE b IS A NATURAL;

WHERE a,c ARE SETS (OF NATURALS);

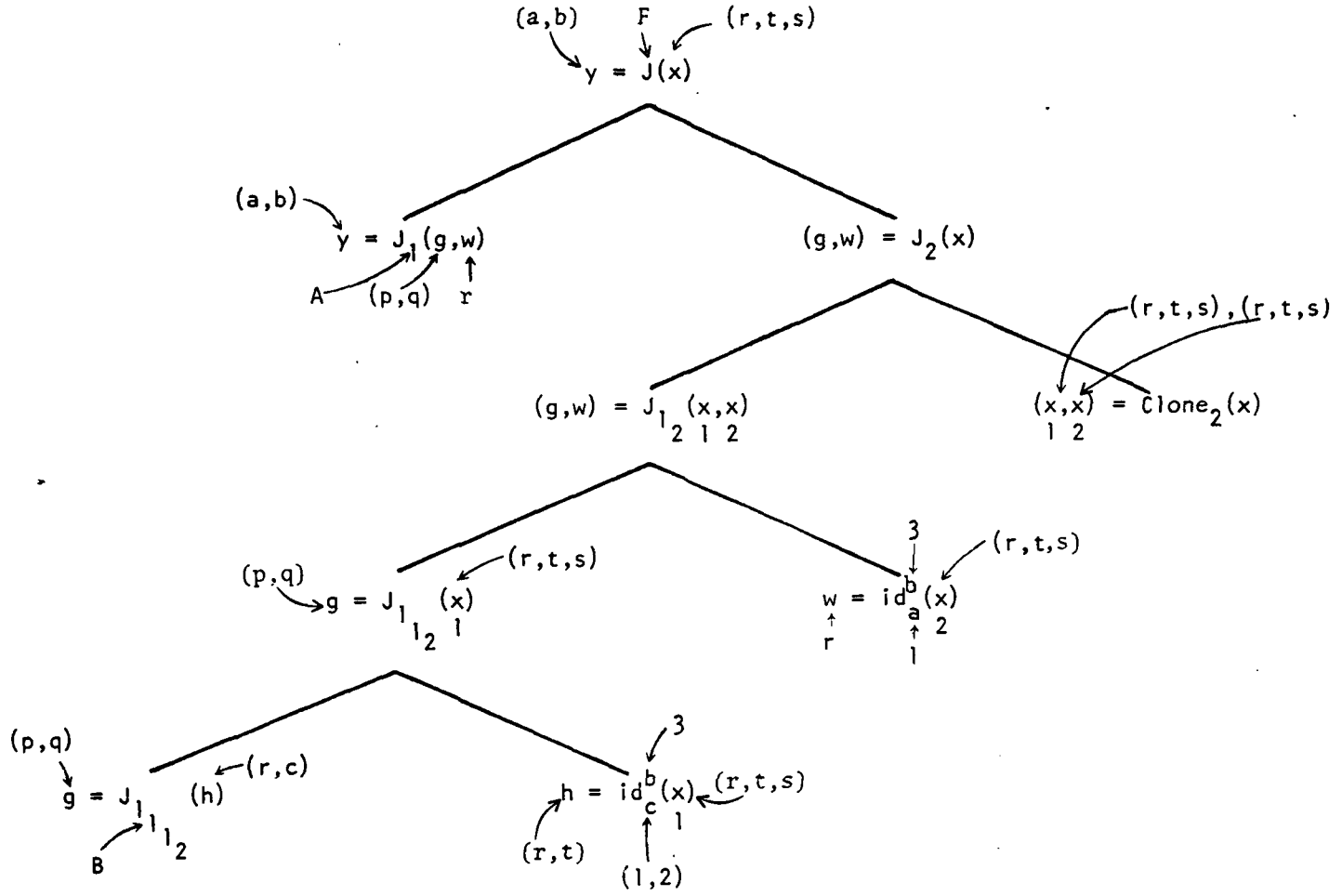


Figure 14

An Instance of the COJOIN Structure

```
WHERE x,y,g,h ARE OF SOME TYPE;

(y1,y2) = COIN1(x,x) JOIN (x,x) = Clone2(x);
          1 2              1 2

y1 = COIN1 (x) INCLUDE y2 = COIN2 (x);
     1 1                    1 2

y1 = COIN1  (g) JOIN g = id^b_a(x);
     1 1                    1

y2 = COIN1  (h) JOIN h = id^b_c(x);
     2 1                    2

SYNTAX:  y1 = COIN1  (g) COINCLUDE y2 = COIN1  (h);
               1 1                       2 1

END COIN;


STRUCTURE:  y = E(x);

WHERE x,y,h,g ARE OF SOME TYPE;

WHERE b IS A NATURAL;

WHERE c,a ARE SETS (OF NATURALS);

y = E1(1x) OTHERWISE y = E2(2x);

PARTITION OF x IS ANY PARTITION;

y = E1  (g) JOIN g = id^b_a(1x);
    1

y = E1  (h) JOIN h = id^b_c(2x);
    2

SYNTAX:  y = E1  (g) COEITHER y = E1  (h);
             1                    2

END E;
```

Structures, in addition to the primitive HOS structures (e.g., COJOIN), can be used to define other structures. For example, the WHEREBY, defined with the COJOIN, gives us the facility to use constant symbols as operands of a function.

STRUCTURE: $y = W(x)$;

WHERE y,h,g,x ARE OF SOME TYPE;

WHERE a IS A SET (OF NATURALS);

WHERE b IS A NATURAL;

$y = W_{1_1}(h,g)$ COJOIN $h = id_a^b(x)$ COJOIN $g = K_{CON}(x)$;

SYNTAX: WHEREBY $y = W_{1_1}(h,CON)$;

END W;

The WHEREBY is used as in

$$y = X+1$$

Here, the constant symbol "1" defines the particular K_{CON} operation that makes the instance of the WHEREBY structure a function. Note also that the operator "+" is used as an infix operator. In AXES we are free to use either prefix or infix notation, as desired.

Another example of a non-primitive structure, XCH, gives the capability to exchange the input values of a function.

STRUCTURE: $(y_1,y_2) = XCH(x_1,x_2)$;

WHERE y_1,y_2,x_1,x_2 ARE OF SOME TYPE;

$y_1 = id_2^2(x_1,x_2)$ COINCLUDE $y_2 = id_1^2(x_1,x_2)$;

SYNTAX: $(y_1,y_2) = XCH(x_1,x_2)$

END XCH;

Often, in a system specification, more than two values of the same variable are desired. For this case, we define the universal operation $y = Clone_n(x)$:

STRUCTURE: $y = C(x)$;

WHERE (y_1, y_2) REPLACE y;

WHERE x,y,g,h ARE OF SOME TYPE;

WHERE m,n,p ARE NATURALS;

$y = C_1(x,m)$ COJOIN $m = K_n(x)$;

$y = C_{1_1}{}^{1}(x,m)$ EITHER $C_{2_1}{}^{2}(x,m)$ EITHER $y = K_{REJECT}{}^{3}(x,m)$;

PARTITION OF (x_1,m) IS

$^{1}(x,m) \mid m = 1$

$^{2}(x,m) \mid m > 1$

$^{3}(x,m) \mid m = 0$

$y = Clone_1(g)$ JOIN $g = id_1^{2\ 1}(x,m)$;

$y_1 = Clone_1(^{2}x)$ COINCLUDE $y_2 = C_{2_{2_1}}{}^{2}(x,m)$;

$y_2 = C_1(x,p)$ COJOIN WHEREBY $p = m-1$;

SYNTAX: $y = Clone_n(x)$;

END C;

In this statement, the only unspecified function (other than the top node) is $m = K_n(x)$. When this structure is used, a value for "n" defines a particular K_n operation which, in turn, is used to define the number of components of "y", each component having the same value of "x".

We can visualize the instance of a structure as being either written down on a piece of paper, by a human being, or to a register by a software or hardware process. To check an instance in an HOS system, the use of a STRUCTURE is compared with the STRUCTURE definition itself, by an analyzer. All instances of a STRUCTURE can be viewed as being supplied to the structure dynamically. The STRUCTURE for an asynchronous machine system, such as an OS or the Higher Order Machine (HOM), described in Section V, is a recursive relation relating each state of a

machine to a previous state of the same machine within an instance of a machine system. To check the instances of an asynchronous machine in a real-time environment, an analyzer is used to not only check the use of the *STRUCTURE* with the *STRUCTURE definition, itself, but also to* check to see that all the users of that STRUCTURE are consistent.

STRUCTURE: $y = HOM(x)$;

WHERE B,A ARE NODAL FAMILIES;

WHERE $HOM_{1_{2_2}}$ IS A CONSTANT FUNCTION;

WHERE K_A IS A FUNCTION;

WHERE x,z,y ARE OF SOME TYPE;

$y = HOM_1{}^1(x)$ OTHERWISE $y = HOM_2{}^2(x)$;

PARTITION OF (x,y) IS

$^1(x,y)|x = REJECT$,

$^1(x,y)|x \neq REJECT$;

$y = HOM(z)$ JOIN $z = HOM_{2_2}(^2x)$;

$z = HOM_{1_{2_2}}(^2x,B)$ COJOIN $B = K_A(^2x)$;

SYNTAX: WHERE A ON HOM;

We indicate the potential happening of each machine instance by specifying a user system to be "ON" the machine system, e.g., the syntax WHERE A ON HOM specifies the initial nodal family of system A to be used by the first machine instance and the nodal family for each next recursive instance of the HOM function K_A to be determined by the ordering relationships of the nodal families within system A. A nodal family is a 3-tuple whose members are functions which stand in a particular relation (c.f. Appendix). By indicating only "HOM" in the syntax of this structure, rather than, for example, "y = HOM(x)," the state of the HOM remains hidden from the user.

9.0 SUMMARY

With respect to the requirements for an operating system, stated earlier in this report, the AGC operating system fulfills some of these requirements (e.g., its data is hidden), but does not fulfill others (e.g., timing is not hidden).

When we began this effort, we thought there was little in the AGC operating system we could improve upon. This attitude was as a result of comparing the AGC with other operating systems, the simplicity of the algorithms in the AGC operating system and the fact that no errors occurred in several years of development in the actual operating system itself. Upon looking back, however, we can now see that many of the development errors which occurred in the user's environment would not have occurred if the AGC operating system had the additional advantages which we discussed above.

Appendix

A FORMAL OUTLINE OF HOS

In HOS, the decomposition process for a system results in a tree structure. At the start of the decomposition process, the entire system is represented by the root of the tree which hopefully, represents the requirements for the system. This definition, however, has many implicit (hidden) requirements. In order to arrive explicitly at the complete definition of the system, the root is decomposed by replacing it with a nodal family (a particular parent node and all of its offspring), which represents the decomposition of the root. This decomposition process, that of replacing a function by its nodal family, can be continued until the entire system has been specified. The resulting tree represents the complete system specification where the leaves represent primitive operations on the data types represented by the variables at those leaves. It may turn out that during the decomposition process a requirement is shown to be erroneous or missing. In such a case, an iteration of the system description is required.

The parent node of the nodal family controls its offspring. When referring to this control relationship, the parent node will be called a module, and its offspring will be called functions. The offspring of the nodal family are the functions required to perform the module's corresponding function (MCF)(i.e., the function that the nodal family replaces).

In the sections that follow, the variable that represents the domain elements of a function is referred to as the input variable, and the variable that represents the range elements of a function is referred to as the output variable. Individual domain and range elements may be called inputs and outputs, respectively.

A module, in performing its corresponding function (Figure A -1), is responsible for determining if the inputs received are in the intended domain of the MCF. If an input is not in the intended domain of the MCF, it is in the unintended domain of the MCF and maps to a special value which is a value of every data type, the value REJECT.

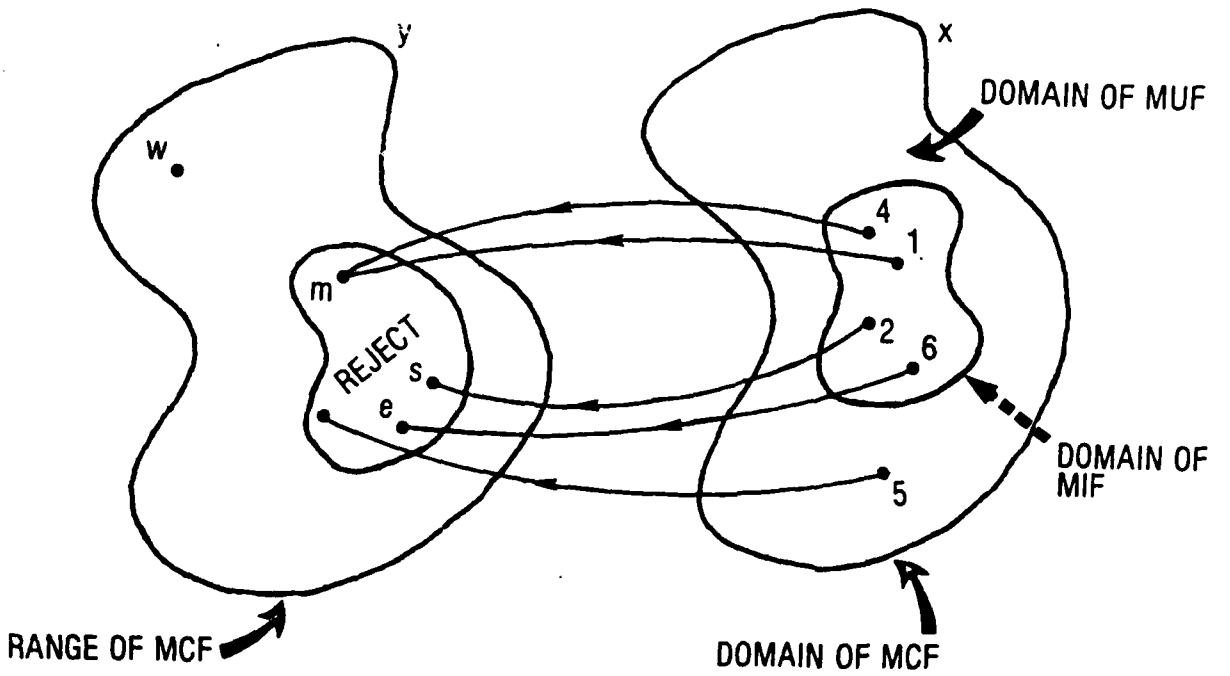


Figure A -1. Illustration of a Function from X into Y

In a sense, the improper input element is not in the domain of the module's corresponding intended function (MIF), but is in the domain of the MCF, i.e., the module's corresponding unintended function (MUF).

Properties of the Primitive Control Structures

While a function can be decomposed in many ways, the HOS axioms [12] provide rules for the construction of nodal families (i.e., the decomposition of a function). From these axioms, three primitive control structures are derived which are used for functional decomposition [16].

These control structures are: composition, set partition, and class partition.

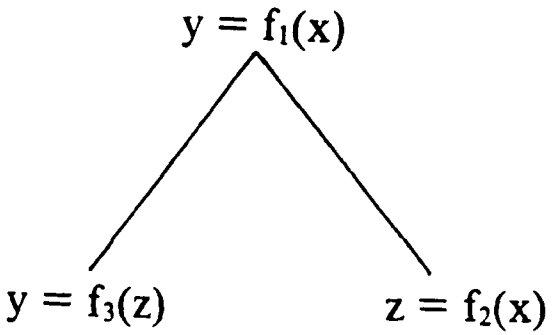


Figure A -2. An Example of Composition

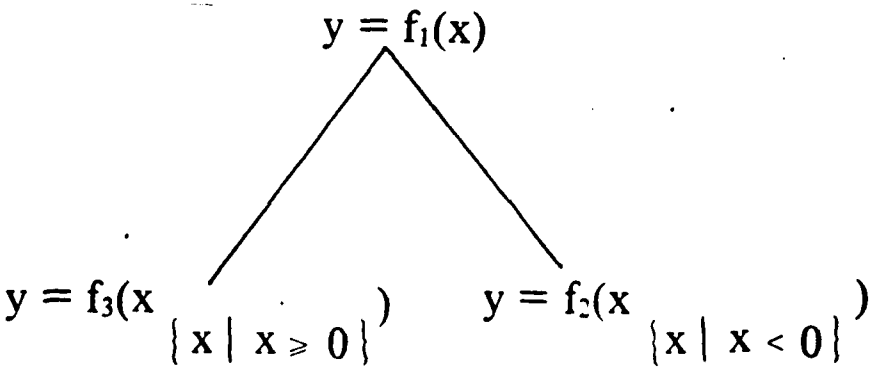


Figure A -3. An Example of Set Partition

Composition is illustrated in Figure A -2. In order to perform $f_1(x)$, the function f_2 must first be applied to x which results in output z. z then becomes an input to f_3 which produces the desired range element of the overall function f_1.

It is important to observe the following characteristics of composition (characteristics are explained with respect to the example in Figure A -2):

(1) One and only offspring (specifically f_2 in this example) receives access rights to the input data, x, from module f_1

(2) One and only one offspring (specifically f_3 in this example) has access rights to deliver the output data, y, for module f_1

(3) All other input and output data that will be produced by offspring controlled by f_1 will reside in local variables (specifically "z" in this example). Local variable, "z", provides communication between the offspring f_2 and f_3

(4) Every offspring is specified to be invoked once and only once in each process of performing its parent's MCF

(5) Every local variable must exist both as an input variable for one and only one function and as an output variable for one and only one different function on the same level.

Set partition, which involves partitioning of the domain, is illustrated in Figure A -3. In the example, the set which comprises the domain is partitioned* into two subsets. For set partition, only one of the offspring will be invoked for each performance of the MCF at f_1 (the determination being based on the value of "x" received) and that offspring will produce the required range element for its parent module when it is performing.

The following characteristics with respect to set partition should be observed:

*Partitioning implies the subdivision of the original set into non-overlapping (i.e., mutually exclusive) subsets.

(1) Every offspring of the module at f_1 is granted permission to produce output values of "y"

(2) All offspring of the module at f_1 are granted permission to receive input values from the variable "x"

(3) Only one offspring is specified to be invoked per input value received for each process of performing its parent's MCF

(4) The values represented by the input variables of an offspring's function comprise a proper subset of the domain of the function of the parent module

(5) There is no communication between offspring

Class partition is illustrated in Figure A -4. While set partition involves partition of the domain into subsets, class partition involves partition of the domain variables into classes and the partition of the range variables into classes. In the example, it is assumed that the domain variable has an associated data structure comprised of two parts, "x_1" and "x_2". Likewise, the range variable has an associated data structure with the same number of classes as the domain's data structure.

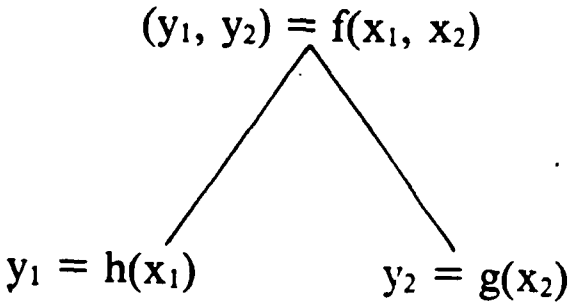


Figure A -4. An Example of Class Partition

The following characteristics with respect to class partition should be observed:

(1) All offspring of the module at f are granted permission to receive input values taken from a partitioned variable in the set of the parent MCF domain variables, such that each offspring's set of input variables are non-overlapping and all the offspring input variables collectively represent only its parent's MCF input variables

(2) All offspring of the module at f are granted permission to produce output values for a partitioned variable in the set of the parent MCF range variables, such that each offspring's set of output variables is non-overlapping and all the offspring's set of output variables collectively represent the parent MCF variables

(3) Each offspring is specified to be invoked per input value received for each process of performing its parent's MCF

(4) There is no communication between offspring.

REFERENCES

[1] Hoare, C.A.R. "Operating Systems: Their Purpose, Objectives, Functions and Scope", Operating Systems Techniques, ed. by C.A.R. Hoare and R.H. Perrott. Academic Press, 1972.

[2] Punj, D., et al. "A Survey of Navy Tactical Computer Applications and Executives." Center for Information Systems Research Report-19, Oct. 1975.

[3] Newman, B., et al. "A Representative Design for a Real-Time Tactical Executive." CENTACS Report No. 57, U.S. Army Electronics Command, Sept. 1975.

[4] Hamilton, M. and Zeldin, S. "The Manager as an Abstract Systems Engineer." Technical Report #5. Higher Order Software, Inc., Cambridge, MA, June 1977. (To be presented at the COMPCON 77 Fall Conference, conducted by the IEEE Computer Society, Washington D.C., Sept. 1977.)

[5] Lickly, D.J., et al. "HAL/S Language Specification." Intermetrics Inc., Cambridge, MA.

[6] Muntz, C. Users Guide to the Block II AGC/LGC Interpreter. Draper Laboratory/MIT Doc. R-489, April 1965.

[7] Hamilton, M. "AGC Program Sundisk." Display Interface Rev. 267, NASA 2021108-011, Draper Laboratory, Cambridge, MA, Nov. 1967.

[8] Lickly, D. "AGC Program Sundisk." Restarts Rev. 267, NASA 2021108-011, Draper Laboratory, Cambridge, MA, Nov. 1967.

[9] Hamilton, M. "Management of Apollo Programming and Its Application to the Shuttle." Software Shuttle Memo No. 29. Draper Laboratory, Cambridge, MA, May 1971.

[10] Dijkstra, E.W. "The Structure of the 'THE' Multi-Programming System". CACM 11, 5, May 1968.

[11] Robinson, L., et al. "A Formal Methodology for the Design of Operating System Software." Computer Science Group, Stanford Research Institute, Menlo Park, CA, Sept. 1975.

[12] Hamilton, M. and Zeldin, S. "Higher Order Software--A Methodology for Defining Software." IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, March 1976.

[13] "Apollo Guidance Program Symbolic Listing Information for Block 2". Revision 1, NAS 9-4810, 27 June 1968.

[14] Van Nostrand, A.D., et al. Functional Writing. Center for Research in Writing, Providence R.I., 1976.

[15] Hamilton, M. and Zeldin, S. "AXES Syntax Description." Technical Report #4. Higher Order Software, Inc., Cambridge, MA, Dec. 1976.

[16] Hamilton, M. and Zeldin, S. "The Foundations for AXES: A Specification Language Based on Completeness of Control." Doc. R-964. Draper Laboratory, Cambridge, MA, March 1976.

[17] Searle, J.R. "Review of J.M. Sadock, Toward a Linguistic Theory of Speech Acts." Language 52, 1976.

Section II

THE SOFTWARE SECURITY PROBLEM AND HOW TO SOLVE IT

by

S. Cushing

TABLE OF CONTENTS

FIGURES

THE SOFTWARE SECURITY PROBLEM AND HOW TO SOLVE IT

So if a man's wit be wandering,
let him study the mathematics

- Francis Bacon

1. SECURITY AND RELIABILITY

When digital computers first began being used in the 1950's, people just programmed their computers in machine or assembly language and ran their programs. With the introduction of higher-order languages, however, and particularly with the development of large and very large software systems, such as those of the Apollo project, for example, a whole new set of questions and problems arose that the early programmers could never have imagined. How can we prevent timing conflicts? How can we prevent data conflicts? How can we prove programs correct? What is the relation between synchronous and asynchronous processing? How can we make an operating system secure? All of these questions and others constitute what Parnas [Par72a] has termed "the so-called 'software engineering' problem" (p. 330).

One of the most interesting instances of this software-engineering problem is that of guaranteeing system security. How can access to the various components of a system be restricted specifically to those for whom it is intended? Linden [Lind76] points out that there are many similarities between the requirements of security and the requirements of reliability, suggesting that "a technical breakthrough on both the security and software reliability problems appears to be as feasible as a breakthrough on the security problem alone" (p. 410). Guaranteeing security requires that "operating systems must be structured so that interactions between system modules are more clearly defined and more closely controlled" (p. 411), but "this same control over the interaction of modules is also needed for reliability."

HIGHER ORDER SOFTWARE, INC. • 843 MASSACHUSETTS AVENUE • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 661-8900

Similarly, "the protection mechanisms needed for security can also be used to enforce software modularity," and "such modularity would improve the reliability and correctness of software." In a word, "there is enough overlap between the requirements for security and the requirements for high system availability that it is reasonable to attempt to solve both problems at the same time." (Availability is a necessary part of reliability, for Linden.)

In this report we will argue that Linden is correct, by showing that software specified according to the Higher Order Software (HOS) methodology of Hamilton and Zeldin [Ham76a,b,77] is automatically secure. HOS was developed as a means of guaranteeing system reliability, without any concern for the security problem per se. Systems specified in HOS are guaranteed against ever having timing or data conflicts [Ham76b]. The fact that they also turn out to be secure makes HOS exactly the kind of common breakthrough that Linden suggests is feasible.

HOS manages to solve these two problems by showing that they need not arise in the first place. If software is specified according to the principles of HOS, then there is no need to ask how to prevent data or timing conflicts, because there simply will be no such thing. Similarly, ignoring history for the moment, if software had always been specified according to HOS, then it would never have occurred to anyone to ask how to make a software system secure, because it simply would have been secure already. Demonstrating this latter point is the purpose of our present paper.

Many people have recognized that the key to solving these problems is to make a clean separation between the specification of a system and its implementation, and, as we will see, HOS is a systems theory that really manages to do this successfully. We will see that trying to solve the reliability, security, and related problems entirely in terms of implementation is like trying to get to the moon on a skateboard. Some systems theories

HIGHER ORDER SOFTWARE, INC. · 843 MASSACHUSETTS AVENUE · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 661-8900

enable us to get off the ground, but then we are stranded forever in the orbit of implementation. HOS enables us, finally, to achieve escape velocity, break free of this orbit, and reach whatever destination we have decided on.

HIGHER ORDER SOFTWARE, INC. • 843 MASSACHUSETTS AVENUE • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 661-8900

2. THE SECURITY PROBLEM

Linden [Lind76] presents a general abstract characterization of system security in terms of what he calls a protection model. Such a model "views the computer as a set of active entities called subjects and a set of passive entities called objects. The protection model defines the access rights of each subject to each object" (p. 415).

Linden represents a protection model in the form of a protection matrix, such as the one in Figure 1 [Lind76, p.416]. The rows of a protection matrix are associated with the subjects of the model and its columns are associated with the objects. "For each subject/object pair, the corresponding entry in the matrix defines the set of access rights that the subject has to the object." For the protection model represented by the protection matrix in Figure 1, for example, we see that subject C may read or execute object X, because both "READ" and "EXECUTE" appear in the matrix slot that occurs at the intersection of row C and column X.

Changes to the protection matrix itself are also controlled by the access rights represented in the matrix; "for example, a subject with 'delete' access to an object can eliminate that object from the protection matrix." Subjects can be allowed to have access rights to each other by having subjects appear also as objects in the protection matrix. "For example, one subject may be allowed to transfer control to another subject by using an 'enter' access right to the other subject."

Linden also introduces the notion of a protection environment, which includes "everything that a subject might cause to be done on its behalf by another subject," as well as everything the subject is allowed to do directly. "A protection domain is a more restricted concept and includes only access rights to objects that are accessible by the subject." The rows of the

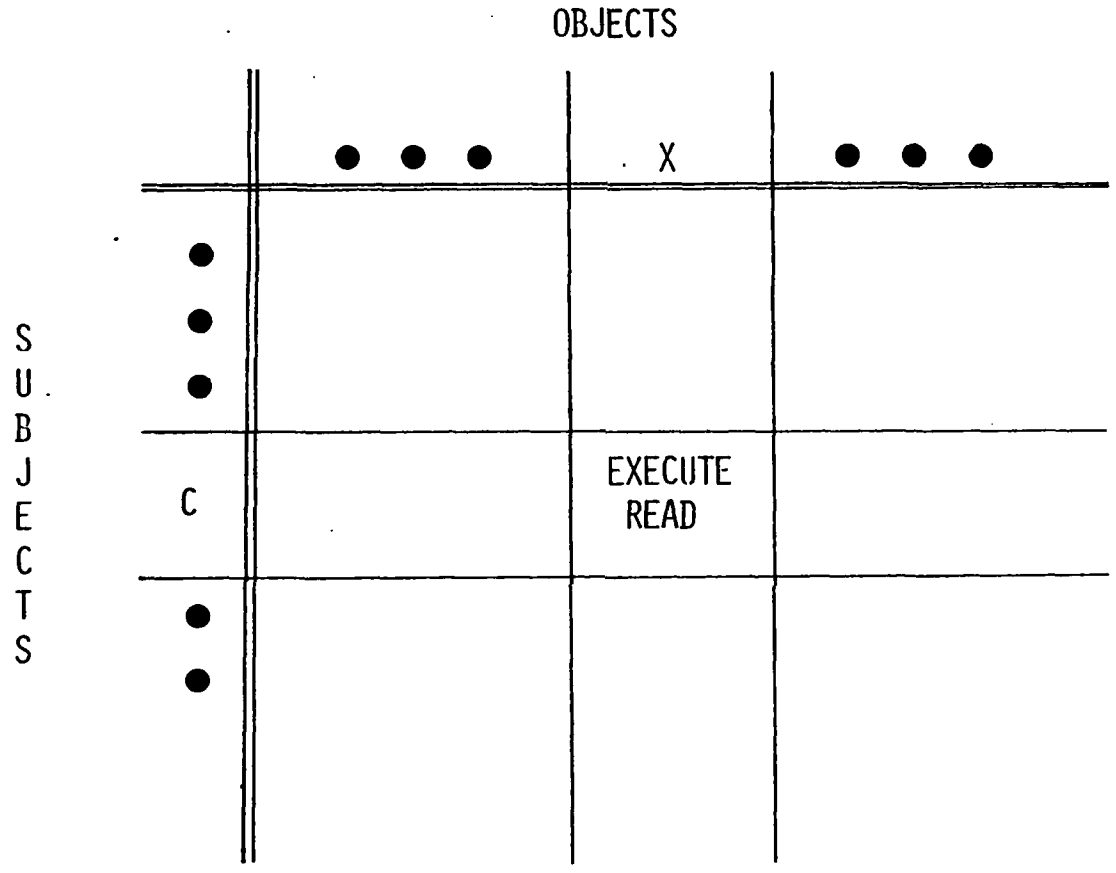


Figure 1: A Protection Matrix [Lind76, p. 416]

HIGHER ORDER SOFTWARE, INC. • 843 MASSACHUSETTS AVENUE • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 661-8900

protection matrix represent the protection domains of the protection model.

The key to Linden's approach to system security is the notion of small protection domains. Linden uses the term "small protection domains" as "a qualitative description of a certain class of protection models. The word 'small' is not intended in a rigid quantitative sense" (p. 416). A small protection domain, for Linden, is the minimal protection domain that will still allow its subject access to everything it has to access. A protection domain may be very large in a quantitative sense, but it is a "small" protection domain if it could not be decreased in size without overly restricting the access rights of its subject. Linden calls this the "principle of least privilege."

Since "a large program usually needs access to many objects," it follows that "protection domains can be kept small only if a large program executes in many different protection domains and constantly switches between these protection domains during its execution." Protection domains can be kept small, "if small subunits of a program execute in their own protection domains," because "a small subunit of a program typically only needs access to a small number of objects." It follows that "the flexibility, ease, and efficiency of domain switching is the primary factor in determining whether protection domains can be kept small and closely tailored to actual needs."

Linden integrates protection domain switching with the calling of a procedure. This permits each procedure to have its own protection domain, even though a domain switch might not be involved in every procedure. A protected procedure, for Linden, is a procedure that does involve a domain switch.

If a procedure is a protected procedure, then it will have a particular protection domain associated with it. "Thus the

right to access certain objects may be available during the execution of that procedure--and possibly only during executions of that procedure." Each execution of a protected procedure will possess the access rights of the procedure, whatever the calling environment may be. The procedure itself, moreover, "can have a state which is preserved between calls to the procedure--and that state is independent of the calling environments."

Linden points out that a protected procedure will appear both as a subject and as an object, when represented in a protection matrix. A protected procedure is an object because there may be other subjects that have the right to call it. This right is represented in a protection matrix by the appearance of a special access right, such as the "enter" access right referred to earlier. A protected procedure also occurs as a subject in a protection matrix because, naturally, "it executes in its own protection domain."

Switching protection domains involves calling a protected procedure. The simplest case of domain switching is the one in which no access rights are passed as parameters in the call. The call takes place and execution begins in the protection domain of the called procedure, as long, of course, as the caller has the right to call this procedure in the first place. Return to the previous protection domain, i.e., the protection domain of the caller, is triggered by a return instruction in the executing called procedure.

This situation is illustrated in the protection matrix in Figure 2 [Lind76, p.417]. User A can call the editor, while executing in his own protection domain. He can also read or write files X and Y either from his own domain or by calling the editor, which is also allowed to read or write files X and Y. The user can use the dictionary, however, only by calling the editor, because the editor, but not the user himself, is allowed to read the dictionary.

OBJECTS / SUBJECTS	EDITOR	FILE X	FILE Y	DICTIONARY
• • •				
USER A	ENTER	READ WRITE	READ WRITE	
EDITOR		READ WRITE	READ WRITE	READ
• •				

Figure 2: Simple Domain Switch

A domain switch is more complex if it involves the passing of access rights to objects as parameters "and if the protected procedure is to be reentrant." This kind of call to a protected procedure creates a new protection domain, i.e., a new row in the protection matrix. "The new protection domain contains both the permanent access rights of the protected procedure," defined by a template domain associated with the procedure, "and the access rights that are passed as parameters in the call."

This kind of situation is illustrated in Figure 3 [Lind76, p.418]. Figure 3a shows the User A's own basic domain and the template domain of the editor. User A has the same access rights as he has in Figure 2, but the editor is allowed only READ access to the dictionary. It cannot read or write files X or Y, as it can in Figure 2. If the user wants to use the editor to read file X, however, he can pass access rights for file X to the editor in the process of calling the editor. This results in the creation of a new protection domain, labeled "INSTANCE OF EDITOR" in Figure 3b, in which the editor does have READ access to file X. Linden notes that "other users may be editing other files using other instances of the same editor."

K. G. Walter [Walt75] presents what is, in effect[1], a formalization of Linden's account of security in the form of a *model for mandatory security*. Walter designs his model to satisfy the "design requirements...that there be no unauthorized disclosure of information and that, otherwise, unrestricted sharing of information be allowed." The model is based on the idea of restricting access to information by giving a specific classification for each piece of information and requiring a user to have the proper clearance in order to access the information.

[1] Calling Walter's characterization of security a formalization of Linden's is probably historically inaccurate, since Walter's account appeared a year and a half earlier than Linden's. This *is* the *logical* relation between the two theories, however, as we show in the text.

objects / subjects	EDITOR	FILE X	FILE Y	DICTIONARY
• • •				
USER A	ENTER	READ WRITE	READ WRITE	
EDITOR TEMPLATE				READ
• •				

Figure 3a: Protection Matrix before Call to Editor

objects / subjects	EDITOR	FILE X	FILE Y	DICTIONARY
• • •				
USER A	ENTER	READ WRITE	READ WRITE	
EDITOR TEMPLATE				READ
INSTANCE OF EDITOR		READ WRITE		READ
• •				

Figure 3b: Protection Matrix during Call to Editor

to access the information.

Formally, Walter describes his model as an 8-tuple

$$M_0 = (R, A, C, \theta, \mu, \trianglelefteq, Cls, Clr)$$

where

R	is a set of repositories.
A	is a set of agents.
C	is a set of security classes.
$\theta \subseteq A \times R$	is the "observe" relation. (a θ r means that agent a can observe the information stored in repository r.)
$\mu \subseteq A \times R$	is the "modify" relation. (a μ r means that agent a can modify the information stored in respository r.)
$\trianglelefteq \subseteq C \times C$	is a pre-ordering of the set of security classes.
CLS: $R \rightarrow C$	is the "classification" function which associates a security class with each repository. (Informally Cls(r) will be referred to as the classification of repository r.)
CLR: $A \rightarrow C$	is the "clearance" function which associates a security class with each agent. (Here again Clr(a) will be referred to as the clearance of agent a.)

Walter's repositories correspond to Linden's objects, while his agents correspond to Linden's subjects. The observe and modify relations correspond to two general kinds of access right that can occur in a protection matrix. The security classes in M_0 correspond to Linden's small protection domains; it is they that determine which repositories

(objects) an agent (subject) can observe or modify (access). There is nothing in Walter's model that guarantees a null intersection of the classes of agents and repositories, so, as with Linden, it is quite possible for some (or all) of the entities involved to be both subjects and objects.

Walter imposes four axioms on his 8-tuple M_0 in order to prove his basic security theorem. The first two axioms state explicitly that the relation $\trianglelefteq$ provides a pre-ordering of the set C of security classes.

Axiom 1: For all $c \in C$, $c \trianglelefteq c$. ($\trianglelefteq$ is reflexive.)

Axiom 2: For all $c, d, e \in C$, $c \trianglelefteq d$ and $d \trianglelefteq e$ implies $c \trianglelefteq e$. ($\trianglelefteq$ is transitive.)

"The second two axioms govern, respectively, the acquisition and dissemination of information."

Axiom 3: For all $a \in A$ and $r \in R$, $a \ \theta \ r$ implies $Cls(r) \trianglelefteq Clr(a)$.

That is, if agent a can observe repository r, then the clearance of a must be greater than or equal to the classification of r).

Axiom 4: For all $a \in A$ and $r \in R$, $a \ \mu \ r$ implies $Clr(a) \trianglelefteq Cls(r)$.

That is, if an agent a can modify repository r, then the clearance of a is less than or equal to the classification of r. Agent a can modify only those repositories with equal or higher security class.)

Walter says that "for making comparisons it is sufficient to assume that the set of security classes is pre-ordered," (p. 286) but his earlier statement that "the classification system has a lattice structure" (p. 286), suggests that he really wants a partial ordering, since it is partial orderings that induce lattice structures. Formally, we include a third ordering

axiom to the effect that something cannot be both higher and lower in the ordering than something else, as follows:

> For all c, d ε C, c $\lhd$ d and d $\lhd$ c implies c = d.

The basic security theorem states that "no information can ever be transferred to a repository in which it can be observed by an agent that does not have sufficient clearance to observe the source repository." Proving this theorem requires the introduction of a "transfer" relation $\tau \subseteq R \times R$, meaning that there is an agent that can transfer information from the first member of R to the second in a particular member of τ. Formally, we say that $\underline{r}\ \tau\ \underline{s}$ for $\underline{r} \varepsilon R$, $\underline{s} \varepsilon R$, if and only if there is an $\underline{a} \varepsilon A$ such that $\underline{a}\ \Theta\ \underline{r}$ and $\underline{a}\ \mu\ \underline{s}$.

The basic security theorem itself requires the reflexive, transitive closure τ^* of τ and the notion of information transfer path. The relation $\underline{r}\ \tau^*\ \underline{s}$ means that "there is a finite sequence of repositories $\{\underline{r}_1\}$ such that $\underline{r} = \underline{r}_1$, $\underline{s} = \underline{r}_{n+1}$, and $\underline{r}_i\ \tau\ \underline{r}_{i+1}$ for all i, $1 \leq i \leq n$." In other words, $\underline{r}\ \tau^*\ \underline{s}$ if and only if information can eventually be passed from $\underline{r}$ to $\underline{s}$. We say that "there is an <u>information transfer path</u> from repository $\underline{r}$ to repository $\underline{s}$," if it is, in fact, the case that $\underline{r}\ \tau^*\ \underline{s}$.

Walter's basic security theorem can be stated formally in either of two ways, as follows:

> Theorem: For all r, s ε R, if r τ^* s, then Cls(r) $\lhd$ Cls(s). In other words, if there is an information transfer path from repository $\underline{r}$ to repository $\underline{s}$ then Cls(r) $\lhd$ Cls($\underline{s}$).
>
> Corollary: If $\underline{r}$ and $\underline{s}$ are repositories and the classification of $\underline{r}$ is not less than or equal to the classification of $\underline{s}$, then there is <u>no</u> information transfer path from $\underline{r}$ to $\underline{s}$.

What this theorem says is that if information flows from one repository to another, then the latter has a security class that is the same as or higher than the former; in other words, information can flow only upwards. Guaranteeing that, in a nutshell, is what the security problem is all about.

3. SPECIFICATION, IMPLEMENTATION, AND LEVELS OF ABSTRACTION

Walter does not stop with M_0, but also presents two other models, M_1, which is outlined in Figure 4, and M_2, which is too complicated simply to exhibit in a figure without further explanation. Walter describes the relationship that is supposed to exist between successive models in the sequence M_0, M_1, M_2 in terms of a "technique of structured modeling" (p. 288), in which successive "levels of modeling" are used to arrive at the full description of a system. He also uses the term "Structured Specification" (p. 285) to denote the approach to specification that results in models that are related in this way. Model M_1 "will satisfy the security requirements in M_0 plus further design requirements... These additional restrictions make the design more implementation specific" (p. 288) by representing the security system as "a file system structured as in a tree of arbitrary depth" and by providing "a mechanism for inter-agent communication which does not require accessing a shared file" (p. 290).

M_2 is a still "more specific security system model" (p. 290) involving "mechanisms which will be used as discretionary controls for access to files." Walter says that the definition of M_0 "has intuitive appeal, however, the way to apply M_0 to a complex operating system is far from obvious" (p. 293). As for M_1, "though still fairly general, this model is appropriate for a small class of machines. The next model, M_2, is applicable to few systems besides Multics," i.e., is getting very close to a description of (part of) an actual operating system, as implemented. "Eventually, some model (probably an M_3 or M_4) will closely resemble commands in the Multics System."

A general framework for understanding what Walter is trying to do is provided by the SRI systems model described by

544 MASSACHUSETTS AVENUE · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 661-8900

M_1 = $(F, M, A, C, \rho_F, \sigma_F, \rho_M, \sigma_M, \trianglelefteq, \delta, Cls, Clr)$

WHERE:

F	is a tree of files
M	is a set of mailboxes
A	is a set of agents
C	is a set of security classes
$\rho_F \subseteq A \times F$	is the "retrieve information" relation. (a ρ_F f means that agent a can retrieve information from file f.)
$\sigma_F \subseteq A \times F$	is the "store information" relation. (a σ_F f means that a can store information in f.)
$\rho_M \subseteq A \times M$	is the "receive" relation. (a ρ_M m means that agent a can receive information through mailbox m.)
$\sigma_M \subseteq A \times M$	is the "send" relation. (a σ_M m means that a can send information to m.)
$\trianglelefteq \subseteq C \times C$	is a pre-ordering of the set of security class.
$\delta \subseteq F \times F$	is the "dominate" relation on the set of files. (It defines the "tree" structure on the files.)
Cls: $F \cup M \rightarrow C$	is the "classification" function for files and mailboxes
Clr: $A \rightarrow C$	is the "clearance" function for agents.

AXIOMS FOR M_1:

A1.1: For all $c \in C$, $c \trianglelefteq c$ ($\trianglelefteq$ is reflexive).

A1.2: For all $c, d, e \in C$, $c \trianglelefteq d$ and $d \trianglelefteq e$ implies $c \trianglelefteq e$ ($\trianglelefteq$ is transitive).

A1.3: For all $a \in A$ and $f \in F$, $a\ \rho_F\ f$ implies $Clr(f) \trianglelefteq Clr(a)$. (An agent can only "retrieve" information from a file with equal or lower classification).

Figure 4: Walter's "Tree Structured Directory Model - M_1"

A1.4: For all $a \in A$ and $m \in M$, $a\ \rho_M\ m$ implies $Cls(m) = Clr(a)$.
(An agent can only "receive" information through a mailbox with classification equal to its own clearance).

A1.5: For all $a \in A$ and $f \in F$, $a\ \sigma_F\ f$ implies $Clr(a) \trianglelefteq Cls(f)$.
(An agent can only "store" information in a file with equal or greater classification).

A1.6: For all $a \in A$ and $m \in M$, $a\ \sigma_M\ m$ implies $Clr(a) \trianglelefteq Cls(m)$.
(An agent can only "send" information through a mailbox with equal or greater classification).

A1.7: For all $f \in F$, $f\ \delta\ f$ (δ is reflexive).

A1.8: For all $f, g \in F$, $f\ \delta\ g$ and $g\ \delta\ f$ implies $f = g$.
(δ is antisymmetric).

A1.9: For all $f, g, h \in F$, $f\ \delta\ g$ and $g\ \delta\ h$ implies $f\ \delta\ h$.
(δ is transitive).

A1.10: For all $f, g, h \in F$, $g\ \delta\ f$ and $h\ \delta\ f$ implies $g\ \delta\ h$ or $h\ \delta\ f$ (sic).
(δ has the "tree" property).

A1.11: For all $a \in A$, and $f, g \in F$, $a\ \rho_F\ g$ and $f\ \delta\ g$ implies $a\ \rho_F\ f$. (In order to retrieve information from a file, an agent must be able to retrieve from (i.e. search) every file which dominates it).

A1.12: For all $a \in A$, and $f, g \in F$, $a\ \sigma_F\ g$ and $f\ \delta\ g$ and $f \neq g$ implies $a\ \rho_F\ f$. (In order to store into a file, an agent must be able to retrieve from or search every file which strictly dominates it. This specifically allows an agent to store in a file from which it cannot retrieve; i.e., write-up is permitted.)

A1.13: For all $a \in A$, and $f, g \in F$, $a\ \sigma_F\ f$ and $f\ \delta\ g$ implies $a\ \sigma_F\ g$. (Since it is expected that attributes of a file will be maintained in a dominating file (directory), if an agent can store into a directory file and thus change attributes of an inferior file, then the agent must also be able to store into (modify) the inferior file).

A1.14: For all $f \in F$, there exists an $a \in A$ such that $a\ \sigma_F\ f$.
(There are no files which cannot be stored into (modified) by at least one agent).

Figure 4: Walter's "Tree Structured Directory Model - M_1" (con't)

Robinson [Robi75], [Robi77].[2] Robinson characterizes a system description in terms of "a sequence of ordered pairs $\{(P_0, M_0), (P_1, M_1), \ldots (P_n, M_n)\}$... called a hierarchically structured program" (p. 272) in which P_i is a set of abstract programs that run on the abstract machine M_i. He notes that, in general, the pairs will occur in a tree structure, and that he assumes a linear ordering only in order to simplify the argument.

> Each program runs on a machine, but since the collection of machines forms a hierarchy, the primitive operations of a machine at some level are realized by a set of programs running on a machine at the next lower level (one program corresponding to each operation of the machine) (p. 272).

"The programs abstract from the implementation details of machines on which they run" and "the only information available to a program is the external behavior of the machine."

The general idea of this structuring is illustrated in Figure 5, in which "M_0 is the most primitive machine and can be viewed as the instruction set for a hardware machine or as a higher-order language" and in which "P_n is the abstract program at the highest level, running on machine M_n." The direction of the arrows in the diagram represent the flow of implementation, in the sense that, "for all values of $i (0 \leq i < n)$, the set of abstract programs P_i running on the abstract machine M_i implements the abstract machine M_{i+1}," while itself running on abstract machine M_i. "The system as a whole is equivalent to some program P running on a machine M, where $M = M_0$ and P_n is an abstraction of P."

Each of the abstract machines in Robinson's framework "can be described as a module of Parnas...in which both the internal state and the transformation rules are characterized

[2] The model description in [Robi75] differs somewhat from that of [Robi77]. We will quote the latter, unless otherwise noted.

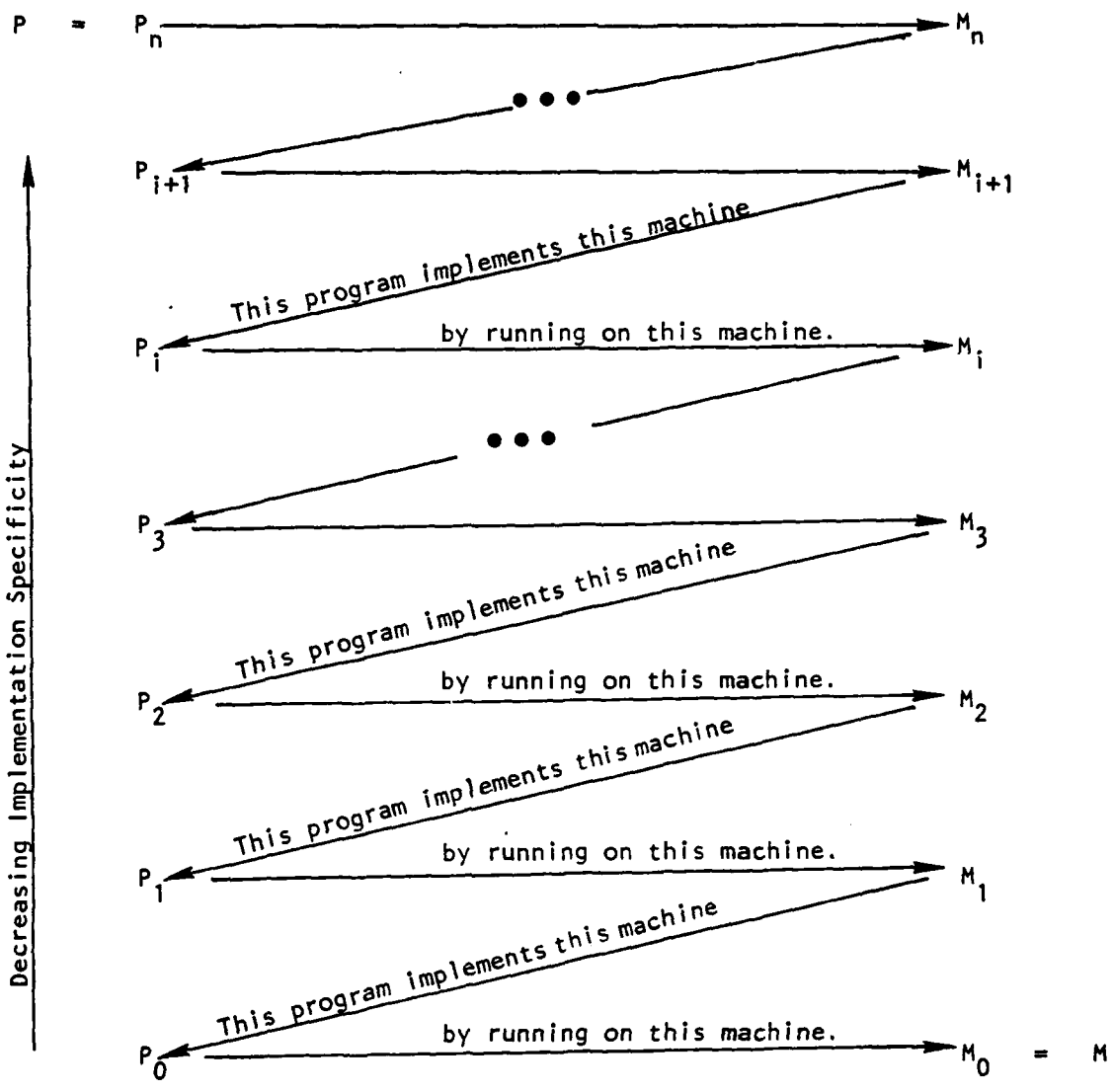


Figure 5

SRI Model of Program P to Run on Machine M

as functions of two types - V-functions (Value functions) and O-functions (Operation functions)." Each "program running on an abstract machine can be expressed as a sequence of calls to the functions that make up an abstract machine." A V-function is one that "returns a value when called; the set of possible V-function values of the module defines the state space (or abstract data structure) of the module." A module's state is denoted by a particular set of values for each V-function. O-functions describe state transformations by defining new values for V-functions. "A state transformation occurs when an O-function is called and is described as an assertion relating new values of V-functions to their values before the call." Such an assertion "is a predicate containing V-functions for which the predicate is true." It "specifies that, as a result of a call, the new state is one of some set of possible states; therefore the specification may be incomplete." The effect of this feature is that it "postpones binding of certain decisions until the abstract program is implemented or even until run-time." An example of an abstract machine characterized as a Parnas module specification is given in Figure 6 [Robi77, p.273].

Except for its reversed numbering scheme, it seems reasonably clear that the SRI framework we have just outlined corresponds more than roughly, in intent, to Walter's "technique of structured modeling" or "Structured Specification." Whereas Walter denotes his most abstract "level of modeling" by the number 0, with increasing numbers as we get closer to implementation, Robinson uses 0 to denote his least abstract "level of abstraction," with numbers increasing as we get further away from that level. The basic idea behind the separation of levels, however, is pretty much the same in both frameworks.

integer V-function: LENGTH

Comment: Returns the number of occupied positions in the register.
Initial value: LENGTH = 0
Exceptions: none

Integer V-function: CHAR(integer i)

Comment: Returns the value of the ith element of the register.
Initial value: ∀i(CHAR(i) = undefined)
Exceptions: I_OUT_OF_BOUNDS: $i \leq 0 \vee i > LENGTH$

O-function: INSERT(integer, i,j)

Comment: Inserts the value j after position i, moving subsequent values one position higher.
Exceptions:

I_OUT_OF_BOUNDS: $i < 0 \vee i > LENGTH$

J_OUT_OF_BOUNDS: $j < 0 \vee j > 255$

TOO_LONG: $LENGTH \geq 1000$

Effects: $LENGTH = 'LENGTH' + 1$

$\forall k(CHAR(k) = \text{if } k \leq i \text{ then } 'CHAR'(k)$
$\text{else if } k = i + 1 \text{ then } j$
$\text{else } 'CHAR'(k-1))$

O-function: DELETE(integer i)

Comment: Deletes the ith element of the register, moving the subsequent values to fill in the gap.

Exceptions: I_OUT_OF_BOUNDS: $i \leq 0 \vee i > LENGTH$

Effects:

$LENGTH = 'LENGTH' - 1$

$\forall k(CHAR(k) = \text{if } k < i \text{ then } 'CHAR'(k)$
$\text{else } 'CHAR'(k + 1))$

Figure 6

Robinson's Register Module Specification

Walter describes the idea behind his methodology as follows:

> In many ways, Structured Specification is similar to Structured Programming; "levels of specification" are analogous to the "levels of abstraction" discussed in *Structured Programming*. *However, in some* sense, these concepts are orthogonal to each other. Structured Programming is a technique for evolving an orderly description of how a particular problem will be solved. Typically, it is a matter of filling in the "nitty-gritty" details of an algorithm which is well understood.
>
> Conversely, Structured Specification concentrates on evolving an orderly description of precisely what problem is to be solved. In addition, the various levels of specification provide a forum for discussing why the program is being designed in a particular way. (p. 285).

Differences in terminology aside (for example, Robinson's "levels of abstraction" would seem to be intended to correspond to Walter's "levels of specification," as well, perhaps, as to the "levels of abstraction" of structured programming), the aim of Robinson's methodology is the same.

Robinson, like Walter, is concerned with specification, not with implementation, except as an ultimate aim. Systems must eventually be implemented, of course, but this is not the point. He describes his methodology as one which "formally represents a program in terms of levels of abstraction, each level of which can be described by a self-contained non-procedural specification." (p. 271). The point is that a program is intended to be characterized in terms of what it is supposed to do (non-procedural), rather than in terms of how (procedural) it is supposed to do it, exactly as Walter says.

Robinson's characterization of a level of abstraction in terms of abstract machines is not a problem, because this involves only a choice of conceptualization and does not necessarily have to affect the formal methodology in an adverse way. A problem is created by the use of abstract programs, however, in the actual characterization of the abstract machines. A program is, by definition, a sequence of instructions, and so is intrinsically procedural. Indeed, Robinson characterizes "a program running on an abstract machine...as a sequence of calls to the functions that make up an abstract machine" (p. 272), as we have seen. As long as a systems framework uses abstract programs to characterize the functions of his primitive machines, we are automatically dealing with the how of those functions, rather than the what, i.e., with their implementation, rather than their specification.

We should note Robinson's assertion that "the Parnas specification language expresses state transformations in a non-procedural way... A Parnas module specification is a self-contained medium for defining an abstraction: V-functions are primitive, and O-functions are described solely in terms of V-functions and the constructs of the assertion language." What he means, presumably, is that, since the O-functions can be reduced to ("described solely in terms of") the V-functions and since the V-functions are primitive, i.e., not further reducible, there is nothing more that he has to do to characterize the module. Those functions (O-) which can be reduced have been reduced and those functions (V-) which have not been reduced need not be reduced, because they cannot be reduced. That, after all, is the meaning of "primitive." While it is true that the primitive elements of a system (any kind of system) cannot (or need not) be further reduced (decomposed, described, etc.) in terms of other elements of the system, however, it by no means follows that there is no need to characterize them at all.

HIGHER ORDER SOFTWARE, INC. • 843 MASSACHUSETTS AVENUE • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 661-8900

Consider a simple case from plane Euclidean geometry. In that geometry, we can take the notions of point and line as primitives and notions like rectangle, triangle, and vertex as non-primitives that can be described in terms of the primitives. Thus point and line correspond to Robinson's V-functions, since he says these are primitive, while rectangle, triangle, and vertex correspond to his O-functions, since he says these are not primitive, but "are described solely in terms of V-functions." A rectangle or a triangle can be described (roughly, to avoid getting too technical and missing the main point) as a particular configuration of lines, and a vertex can be described as a point that is the intersection of two lines. Thus the non-primitives are described in terms of the primitives, exactly as Robinson wants.

The story does not end here, however. While reduction of geometric entities ends at the level of point and line (and perhaps other primitives, which we are ignoring for simplicity), point and line themselves are then characterized in terms of each other, i.e., in terms of their mutual interaction, by means of axioms. Something is a point or a line if and only if it behaves in accord with the axioms. The axioms of a geometry, in fact, are its most important part, because everything else about the geometry follows from them, once the appropriate definitions of non-primitive entities in terms of primitive ones are stated.

What this means in Robinson's case is that it is not enough simply to state that the V-functions are primitive and leave it at that. Looking carefully at Figure 6, we see that the only way that V-functions are characterized within the module is in terms of informal comments, in English, that tell us what the functions are supposed to do. The formalism, however, places no constraints on what these functions can do, except for giving them initial values and (perhaps) restricting their domains. Literally, any function that has these initial

values and these domains can serve as the LENGTH and CHAR functions in the module. Since this is too general for what Robinson intends, he is forced to narrow down the candidate functions for LENGTH and CHAR by characterizing them outside of the module in terms of abstract programs, which do spell out formally and, by definition, algorithmically the functions that he wants. This step, however, ipso facto removes us from the realm of specification and places us in that of implementation. In the process, we lose "the major advantages of Parnas specifications." namely, "that they abstract from the algorithms of implementation and are self-contained" (p. 272).

We see that Parnas' modules do not really characterize their functions completely, as they are supposed to. One of the underlying reasons for this problem is that Parnas tries to make his modules do too much. Parnas confuses the need to decompose a system into subsystems with the need to characterize in precise terms the kinds of objects the system deals with, proposing that both needs can be satisfied with his single notion of module.

In many places, Parnas talks about "dividing the system into modules " [Par72b, p. 1053] and "decomposing a system into modules," so it is clear that modules are intended to be the kind of thing into which systems are decomposed. With respect to the STACK module in Figure 7, however, he tells us that it is proposed as a definition of a kind of object:

> We propose that the definition of a stack shown in Example 1 should replace the usual pictures of implementations (e.g., the array with pointer or the linked list implementations). All that you need to know about a stack in order to use it is specified there. There are countless possible implementations (including a large number of sensible ones). The implementation should be free to vary without changing the using programs. If the using programs assume no more about a stack than is stated above, that will be true. (p. 332)

It follows from these facts that Parnas is decomposing systems into kinds of objects, but this is not the sort of result he really wants. It is this kind of inadequacy that leads Robinson to try to augment the Parnas methodology with things like abstract programs.

```
Function PUSH(a)
possible values:  none
integer:  a
effect:  call ERR1 if a > p2 ∨ a < 0 ∨ 'DEPTH' = p1
     else [VAL = a; DEPTH = 'DEPTH" + 1;]

Function POP
possible values:  none
parameters:  none
effect:  call ERR2 if 'DEPTH' = 0
     the sequence "PUSH(a); POP" has no net effect if no error
        calls occur.

Function VAL
possible values:  integer initial; value undefined
parameters:  none
effect:  error call if 'DEPTH' = 0

Function DEPTH
possible values:  integer; initial value 0
parameters:  none
effect:  none
p1 and p2 are parameters.  p1 is intended to represent the
     maximum depth of the stack and p2 the maximum width or
     maximum size for each item.
```

Figure 7

Parnas' Stack Module

4. HOS AS A GENERAL SYSTEMS THEORY

Like Linden and Walter, HOS recognizes that there are essentially two modes of existence in the world, that of *being* and that of *doing*, and that everything generally manifests both modes at once. A given thing can either *be* or *do* and, in general, will both *be* and *do* at the same time. This dichotomy reflects the related bifurcation between *being* and *becoming*. If there is something that is *doing*, then there is something (perhaps the same thing) that is *being done to*, and this latter thing is therefore *becoming*. Again, in general, anything that is *doing* is also *being done to* and so is itself *becoming*, as well as *being*.

This enables us to understand the important relationship between *constancy* and *change*. If we remove the front element from a queue, for example, we still have the *same* queue, with one element removed, but we also have a *different* queue, i.e., the one that differs from the original one in exactly that element. The queue can still *be* the *same* queue, even though it has *become* a *different* queue, and we are free to choose whichever of these aspects of the situation fits our needs for any particular problem. We can also say the queue has changed its *state*, stipulating that the queue itself has not changed, but then it is the *states* that are *being* or *becoming*, so the same dichotomy emerges again on a higher level of abstraction.

Linden expresses the distinction between *being* and *doing* in terms of his distinction between *objects* and *subjects*, as we have seen. Objects are things that *are done to*, i.e., they simply *are*, rather than *do*. Subjects, in contrast, are things which *do*, and the objects are precisely the things they *do to*. Walter expresses this dichotomy in terms of his distinction between *repositories* and *agents*, as we have also

seen. Agents are things which do, and repositories are things which are and which therefore are done to by the agents. As we have discussed, anything, in general, will both be and do, so anything is both an agent and a repository and both a subject and an object, as Linden, and presumably Walter, would agree.

While both Linden and Walter thus recognize this fundamental dichotomy in any system, there are serious defects in their formulations of this dichotomy. The problem with Linden's formulation is that it is not formal. All he tells us is that "a protection model views the computer as a set of active entities called subjects and a set of passive entities called objects" (p. 415), with no formal characterization of what these subject/object things or their properties are supposed to be. Such an omission is perfectly justifiable in the context of the general survey sort of article in which it occurs, but it must be corrected in a complete systems theory.

Walter's formulation is quite formal, but it falters in a different respect. A fully general systems theory should be capable of expression at the highest possible level of generality. Like Linden's account it should state things solely in terms of subjects and objects, i.e., things that do and things that are, at this highest level of generality, while permitting subcategorizations of these basic categories, e.g., procedure, protection domains, etc., at lower levels of generality. Walter's problem is that he conflates levels by including something not at all on a par with agents and repositories with respect to generality, i.e., security classes, on the highest level of generality of his systems theory. Again, within a sufficiently limited domain of interest, Walter's decision to lump the highly specific notion of security classes in with the completely general notions of agent and repository is excusable, but outside of such a domain,

it will place unnecessary restrictions on any system specified in accordance with the theory. A general systems theory should allow the introduction of lower-level notions like security classes, if they are needed, but it should not require them on its most general level, where only agents and repositories should reside.

HOS expresses the distinction between being and doing in terms of the familiar notions of data and function, and it does this in a completely formal way. Anything that can be can be represented as a member of a data type, and anything that can do can be represented as a function[3]. As we would expect from a correct formulation, anything that can be, i.e., a datum, can also do, by serving as input to a function, and anything that can do, i.e., a function, can also be, since functions themselves make up a data type.

For example, if datum x is mapped by functions f_1, f_2, f_3, f_4, f_5 onto data y_1, y_2, y_3, y_4, y_5, respectively, then x itself can be viewed as a function that maps the data f_1, f_2, f_3, f_4, f_5 onto y_1, y_2, y_3, y_4, y_5. Functions themselves can be data, in other words, and data can be functions, depending on the requirements of the particular problem we are working on. If FXY is the subset of data type FUNCTION whose members map data type X into data type Y, then X is the subset of FUNCTION that maps FXY into Y. Both interpretations are correct, in general, and which one we choose depends on what we need for a specific problem.

In our formulation, however, unlike Linden's, this reversability follows naturally from the nature of data and functions. We do not really have to say explicitly that subjects can also be objects and vice versa, because that fact follows automatically from our identification of subjects with functions and objects with data.

[3] [Ham76a] uses the term "function" in a more highly restricted sense and the term "operation" in the sense of our "function." For our present purposes, the distinction is unimportant, and we will use the two terms interchangeably.

Again in accordance with the fundamental dichotomy, although data and functions are distinct components of systems, they are at the same time inseparable from each other, because each is characterized formally in terms of the other. A function consists of an input data type, called its domain, an output data type, called its range, and a correspondence, called its mapping, between the members of its domain and those of its range; a function can be characterized, therefore, as an ordered triple (Domain, Range, Mapping), where the components are as we have just stated. A data type consists of a set of objects, called its members, and a set of functions, called its primitive operations, which are specified by giving their domains and ranges, at least one of which for each primitive operation must include the data type's own set of members, and a description of the way their mappings interact with one another and, perhaps, with those of other functions; a data type can thus also be characterized as an ordered triple, this time (Set, DR, Axioms), where Set is the set of its members, DR is a statement of the domains and ranges of its primitive operations, and Axioms is a description of the interactive behavior of the mappings of the primitive operations.

An example of an HOS data-type specification, namely, type STACK, is given in Figure 8, written in the HOS specification language AXES [Ham76a]. It is not difficult to see that this specification avoids all of the problems that we discussed in connection with Parnas' stack module in Figure 7. The specification in Figure 8 has absolutely nothing to do, by itself, with system decomposition. It is a definition of a kind of object, plain and simple, and thus serves exactly the kind of purpose it is suited to serve, rather than trying to overextend itself, as Parnas' module does. Furthermore, it is entirely self-contained, because the primitive operations are characterized in terms of each other, rather than being left dangling in the "module" to be rescued by abstract

```
DATA TYPE:  STACK;
PRIMITIVE OPERATIONS:
stack_1 = Push(stack_2, integer_1);
stack_1 = Pop(stack_2);
integer_1 = Top(stack_1);

AXIOMS:
     WHERE Newstack IS A CONSTANT STACK;
     WHERE s IS A STACK;
     WHERE i IS AN INTEGER:
Top(Newstack) = REJECT;
Top(Push(s,i)) = i;
Pop(Newstack) = REJECT;
Pop(Push(s,i)) = s;
END STACK;
```

Figure 8

HOS/AXES Data Type Stack

programs. Finally, it is absolutely implementation-free, because any implementation, whether made up of vacuum tubes, transistors, integrated circuits, magnetic bubbles, or ice-cream cones, will be a satisfactory implementation, as long as primitive operations can be defined in the implementation that behave in accordance with the axioms.

An interesting thing happens when we try to specify Walter's M_0 in terms of HOS data types. The first thing we notice about M_0 is that repositories are more basic than agents. An agent, in Walter's terms, is anything that can observe or modify a repository, while a repository is anything at all that can be partially ordered. Walter says that "associated with each repository is a security class which measures the relative sensitivity of the information stored within it." Since the only real function of the security class is to measure "relative sensitivity," it follows that their function could be accomplished just as well by partially ordering the repositories themselves. This enables us first to characterize the class of repositories as a data type independently of the class of agents and then to characterize the class of agents as a data type in terms of the data type REPOSITORY. It also enables us to eliminate the class of security classes altogether from our model by imposing our partial ordering directly on the data type REPOSITORY and assigning each agent a maximal repository it can observe and a minimal repository it can modify. This confirms our earlier observation that Walter is conflating levels of generality in his model. Security classes can be introduced as a data type at a lower level of generality, if they are really needed for a particular problem, or if they are simply desired for reasons of convenience or elegance, but they have no place on the highest level of generality of a general systems theory.

Figure 9 gives the HOS specification of data type REPOSITORY, written, as usual, in AXES. As just noted, the only primitive operation we need in this data type specification is the partial ordering Atmost, whose axioms are available with AXES and thus do not need to be stated explicitly.[4]

```
DATA TYPE:  REPOSITORY;
PRIMITIVE OPERATIONS:
   boolean = Atmost(repository1, repository2)
AXIOMS:
END REPOSITORY;
```

Figure 9

HOS specification of data type REPOSITORY

Note that whereas Walter treats his partial ordering as a general relation, i.e., as a general subset of C x C, or equivalently, a general set of ordered pairs (C_1,C_2), we treat it as a function, i.e., a subset of REPOSITORY x REPOSITORY x BOOLEAN in which the first two components of each (R_1,R_2,b) uniquely determine the third. The possibility of treating any relation as a function that maps into BOOLEAN is a general property of relations which HOS takes full advantage of. It enables us to integrate the treatment of relations that might not normally be viewed as functions into the general functional-decomposition framework of HOS and thus to see how such "non-functional" relations fit into the system as a whole of which they are a part.

[4]Equality is also needed, but this is provided in AXES itself for every data type. Atmost is not a universal operation, as Equality is, but is universally available, in that we can include it in any data type specification with whose axioms its own axioms are consistent. The axioms of Atmost are stated once and for all in AXES and thus need not be restated every time the operation is included among those of a particular data type. Once Atmost is included among the primitive operations of a particular data type, its axioms are automatically those that are stated for it in the theory. See [Cus77a] for discussion of these ideas.

Figure 10 gives the AXES specification for data type AGENT[5]. As noted earlier, there is one primitive operation, Observeclearance, that assigns to each agent a maximal repository it can observe and a second primitive operation, Modifyclearance, that assigns to each agent a minimal repository it can modify. The remaining two operations, Observes and Modifies, correspond to Walter's θ ("observe") and μ ("modify") relations, respectively, in the way discussed in the preceding paragraph.

```
DATA TYPE:  AGENT;
PRIMITIVE OPERATIONS:

  repository = Observeclearance(agent);
  repository = Modifyclearance(agent);
  boolean = Observes(agent,repository);
  boolean = Modifies(agent,repository);

AXIOMS:
  WHERE a IS AN AGENT
  WHERE r IS A REPOSITORY
(Observes(a,r) ⊃ Atmost(r,Observeclearance(a)) = True;
(Modifies(a,r) ⊃ Atmost(Modifyclearance(a),r)) = True;
Atmost(Observeclearance(a), Modifyclearance(a)) = True;

END AGENT;
```

Figure 10. HOS Specification of Data Type AGENT

The three axioms of data type AGENT together provide the effect of Walter's Axioms 3 and 4, without the use of "security classes." The first axiom says that if an agent can observe a repository, then that repository must be lower (but not necessarily strictly lower) in the partial ordering of repositories than the maximal repository the agent can observe. The axiom functions, in other words, as a mutual definition of "can observe" and "maximal observable repository" in terms of each other and the partial ordering, in the usual manner of HOS data-type axioms. The second axiom says that if an agent can modify a repository, then that repository must be higher (though perhaps not strictly higher) in the partial ordering of repositories than the minimal repository that the agent can modify. This functions, again, as a mutual definition of "can modify" and "minimal modifiable repository" in terms of each other and the partial ordering.

Given the first two axioms, the third axiom provides all of the effect of Walter's "security classes" by guaranteeing that the maximal observable

[5]The symbol "⊃" is a traditional infix symbol for material implication in formal logic and is used here in place of the AXES prefix operation symbol "Entails" [Ham76a]. It seems reasonable to use such traditional infix symbols as abbreviations for AXES prefix symbols, whenever this is convenient, and this convention is adopted explicitly in [Ham76a] and [Cus77a].

repository is always lower in the partial ordering than the minimal modifiable repository. This means that, for a given agent, the lattice of repositories can be divided into an "upper half" and a "lower half," such that the agent can observe only repositories in the lower half and modify only repositories in the upper half. This, however, is really the only purpose that security classes serve in M_0, so we really can dispense with them entirely, as we have done.

In Walter's terminology, we have reduced his 8-tuple

(R, A, C, θ, μ, $\trianglelefteq$, Cls, Clr)

to a 7-tuple

(REPOSITORY, Atmost, AGENT, Observes, Modifies, Observeclearance, Modifyclearance)

by showing that one of his data types is superfluous and that his primitive operations that map into that type can be replaced by different primitive operations which have the same effect but which have only the two remaining data types as domains and ranges. Whereas Walter's 8-tuple requires two special axioms, besides those for the partial ordering, which are intrinsic to AXES, but which Walter has to state, making a real total of five axioms for him, our 7-tuple requires only three explicitly stated axioms, as shown in Figure 5.

It should be noted that if we had tried to specify explicitly all three data types that Walter proposes, we would immediately have run into problems. Walter names his data types and describes how his operations (functions/relations) are supposed to work, but he does not explicitly specify either the operations or the types. His Axioms 3 and 4, for example, really express relationships between types, rather than defining characteristics of the individual types themselves. From the HOS point of view, this amounts to putting the cart before the horse, stating a relationship between two things before we have any idea at all what it is that is being related. From Walter's point of view, of course, this is perfectly legitimate, because, presumably, he views the situation as being

analogous to that of points and lines in plane geometry, which also are usually characterized not independently as data types, but in terms of each other. The advantage of our point of view is its complete generality. Identifying being things and doing things with data (types) and functions, respectively, enables us to specify any system at all in a principled way, without introducing any further kinds of entities. Walter's formulation of this distinction in terms of a mutual definition of repositories and agents, in contrast, still requires him to use functions (and relations, for that matter) to define his repositories and agents. In our framework, repositories and agents are data and functions, respectively, and that is the end of that.

We could have defined a type "SECURITY CLASS" in terms of the partial ordering, for example, but then we would have been unable to write axioms on the data types AGENT and REPOSITORY for the "primitive operations" CLS and CLR that map these types into that type without introducing a host of other "primitive operations." Similarly, there would have been no non-arbitrary way to decide whether θ and μ, which take both agents and repositories as input, should be "primitive operations" on AGENT or on REPOSITORY. By recognizing that the only function of "SECURITY CLASS" in M_0 is to provide an appropriate partial ordering for REPOSITORY, we can see that REPOSITORY is a more basic data type than AGENT and define the partial ordering directly on REPOSITORY, as we did. In other words, REPOSITORY is "SECURITY CLASS" at the level of generality at which M_0 is defined. Whether we call that single type "REPOSITORY" or "SECURITY CLASS" is, of course, entirely a matter of choice.

The other important function that Parnas tries to make his modules serve, i.e., system decomposition, is specified in HOS in terms of decomposition trees, also called control maps. Given a system that involves certain data types, the function the system performs can be decomposed into a tree structure whose nodes are functions and whose terminal nodes, in particular, are primitive operations of the data types,

where the collective effect of the functions at the terminal nodes is the same as that of the system as a whole. Such tree structures are not intended to provide definitions of kinds of objects, as Parnas' modules are, but represent system decompositions into subsystems, plain and simple. An example of such a decomposition tree, for the function $y = \frac{a+b}{c-d}$, is shown in Figure 11. The domain and range of the decomposed function can be determined by the typed variables that represent inputs and outputs and by the primitive operations that appear at the terminal nodes. The tree itself is precisely what gives the mapping of the decomposed function, by showing how that mapping gets accomplished in terms of the collective behavior of the independently characterized primitive operations.

The key to the usefulness of these decomposition trees lies in the six HOS axioms, listed in Figure 12. It is these axioms, in fact, and their consequences, of course, that make HOS HOS. While HOS can specify any system that can be specified, the specification must be in accordance with these axioms or the system may be incomplete or unreliable. Any software system, in particular, that is specified in accordance with these axioms is automatically guaranteed to be reliable, in the sense that no data or timing conflicts can ever occur [Ham76b]. Formally, the axioms tell us that a well-formed HOS tree is always equivalent to a tree in which every node is occupied by one of the three primitive control structures, shown in Figure 13. Abstract control structures, defined in terms of the primitives may also appear in well-formed trees, and, conversely, any control structure, i.e., configuration of parent and offspring nodes, can appear in a well-formed tree as long as it can itself be decomposed into the primitives.

Such an HOS tree can be interpreted either as decomposing a function into primitive operations or as building up a function out of primitive operations. Which interpretation we

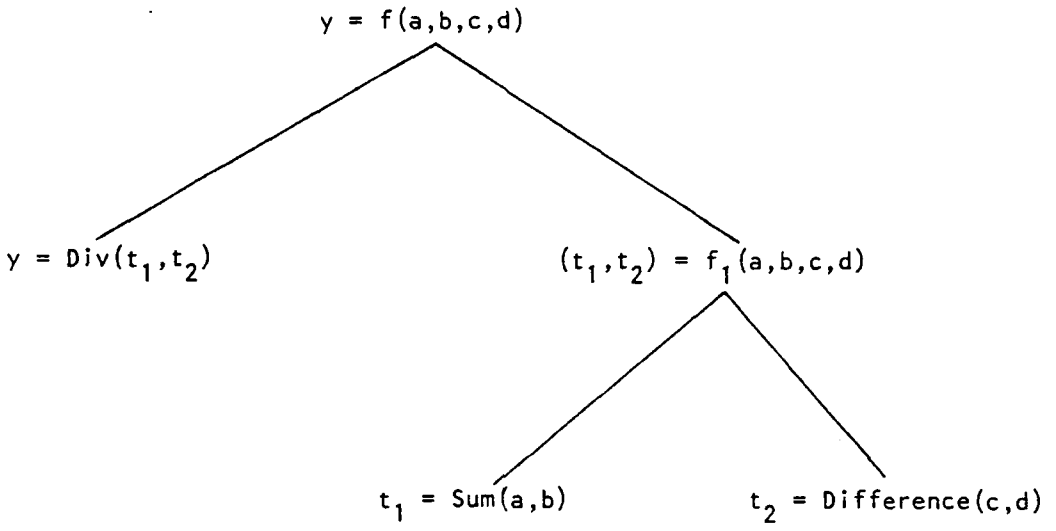


Figure 11

HOS Tree for Function $y = \frac{a+b}{c-d}$

DEFINITION: Invocation *provides for the ability to perform a function.*

AXIOM 1: A given module controls the invocation of the set of functions on its immediate, and only its immediate lower level.

DEFINITION: Responsibility *provides for the ability of a module to produce correct output values.*

AXIOM 2: A given module controls the responsibility for elements of its own and only its own output space.

DEFINITION: *An* output access right *provides for the ability to locate a variable, and once it is located, the ability to give a value to the located variable.*

AXIOM 3: A given module controls the output access rights to each set of variables whose values define the elements of the output space for each immediate, and only each immediate lower-level function.

DEFINITION: *An* input access right *provides for the ability to locate a variable, and once it is located, the ability to reference the value of that variable.*

AXIOM 4: A given module controls the input access rights to each set of variables whose values define the elements of the input space for each immediate, and only each immediate lower-level function.

DEFINITION: Rejection *provides for the ability to recognize an improper input element in that, if a given input element is not acceptable, null output is produced.*

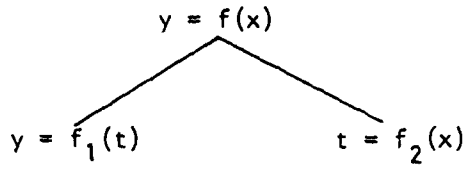
AXIOM 5: A given module controls the rejection of invalid elements of its own, and only its own, input set.

DEFINITION: Ordering *provides for the ability to establish a relation in a set of functions so that any two function elements are comparable in that one of the said elements precedes the other said element.*

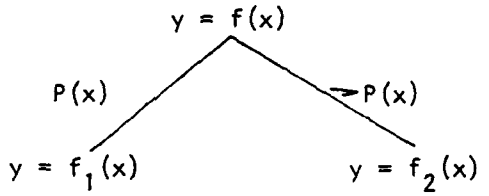
AXIOM 6: A given module controls the ordering of each tree for its immediate, and only its immediate, lower level.

Figure 12

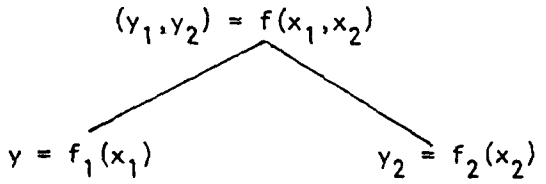
The Axioms of HOS



COMPOSITION



SET PARTITION



CLASS PARTITION

Figure 13

The Three Primitive Control Structures of HOS

choose for a particular tree depends, as usual, on the use we want to make of it. Under either interpretation of such a tree, however, what we end up with is a specification of the function at its root node that is genuinely non-procedural, i.e., non-algorithmic, and entirely free of implementation considerations. The tree provides a complete and explicit account of what functional mapping the function performs and how that mapping is collectively carried out on the types involved by their primitive operations. Everything is clearly spelled out in terms of the hierarchical organization of functional mappings, and this --no more, no less-- is exactly what we require of an adequate specification methodology. The need for abstract programs, i.e., (procedural) sequences of abstract calls to the primitive operations of abstract machines, is entirely eliminated. It follows that replacing each of Robinson's P_i's with an HOS tree will make the problems we found in connection with his "abstract programs" disappear.

It is worth noting, at this point, that HOS does not distinguish at all between O-functions and V-functions, because, however important this distinction may be in particular implementations, it simply does not exist from the point of view of specification, i.e., on the highest level of generality. Functions are things that do, as opposed to be. Sorting out different kinds of functions is something we can do at lower levels of generality, but has no place as a requirement of the theory itself.

To illustrate this point again, suppose we have a register whose positions are filled with integers, as in the example of Figure 6 (a stack or queue would do just as well for our purposes; c.f. Figure 8 for data type stack and [Cus77b] for data type priority queue, for example). Obviously, there is a big difference between an implemented register and the integers it contains, and thus between changing the state of

the register and taking one of those integers as a value. From the point of view of specification, however, a register is every bit as much of an abstraction as an integer. The two abstractions differ, moreover, only in the interactive behavior of the primitive operations that are used to characterize their data types, as this behavior is specified in the axioms of the respective type. From the point of view of specification, therefore, changing the state of an implemented register amounts simply to producing a new abstract register as a value. If we take a register and remove its last element, for example, we get a new register that is identical to the original register except that it lacks the original register's last element. This may not be what happens in implementation, but it is the logic of the situation, and that is what specification is really all about[6].

As we observed earlier, Robinson supplements his "abstract machines" with "abstract programs" in order to do fully the two jobs that Parnas wants his modules to do. Robinson's "abstract programs" tell us what the functions really are that are intended to be characterized in the modules.

Robinson's intention can be successfully achieved by replacing each component of his framework with a corresponding component of HOS. Since his "abstract programs" serve as the characterizations of functions, we replace each of them with a decomposition tree. This relieves his "abstract machine" modules of the burden of serving as the units of system decomposition and leaves them free to serve as definitions of kinds of objects, which is what they would prefer to do anyway, as we have seen. We thus replace each of the "abstract machines" with a set of data-type specifications of HOS.

---

[6] Note that this is just another way of looking at what we said about queues in the second paragraph of this section.

Formally, then, we replace each of Robinson's ordered (P,M) pairs with an ordered pair (D,T), where D is a set of data types replacing the "abstract machine" M and T is a set of decomposition trees replacing the set of "abstract programs" P. Robinson's levels of abstraction gets replaced with a data level of HOS. For simplicity, we will assume that the data levels are linearly ordered, in order to preserve the analogy that we are developing with Robinson"s account of the SRI methodology, but, in fact, only a partial ordering is really necessary, as long as there is a maximal data level in the ordering that contains only one tree.

Higher data levels are related to lower data levels in that the composition trees of each data level decompose the primitive operations of the next higher data level in terms of the primitive operations of the lower data level[7]. For every primitive operation f of a member of D_{i+1} ($i \geq 0$), in other words, there will be a decomposition tree in T_i whose root is f and whose leaf nodes are primitive operations of a member of D_i. The primitive operations of the lowest-data level data types D_0 are the primitive operations of the system, because these are not decomposed at all, but are characterized only in terms of their axiomatic interaction. The D_i thus play the role of Robinson's M_i and the T_i play the role of his P_i, as we said we want them to do, but avoiding any suggestion of implementation.

The simplest case, in which each D_i contains a single data type and in which T_n contains only one decomposed function f, corresponding to Robinson's single program P, is illustrated in Figure 14 which clearly reveals the parallel between the HOS

---

[7]We are restricting our discussion of HOS here somewhat, in order to maintain as close an analogy as possible with Robinson's framework. Later we will expand our account by discussing HOS in fuller generality.

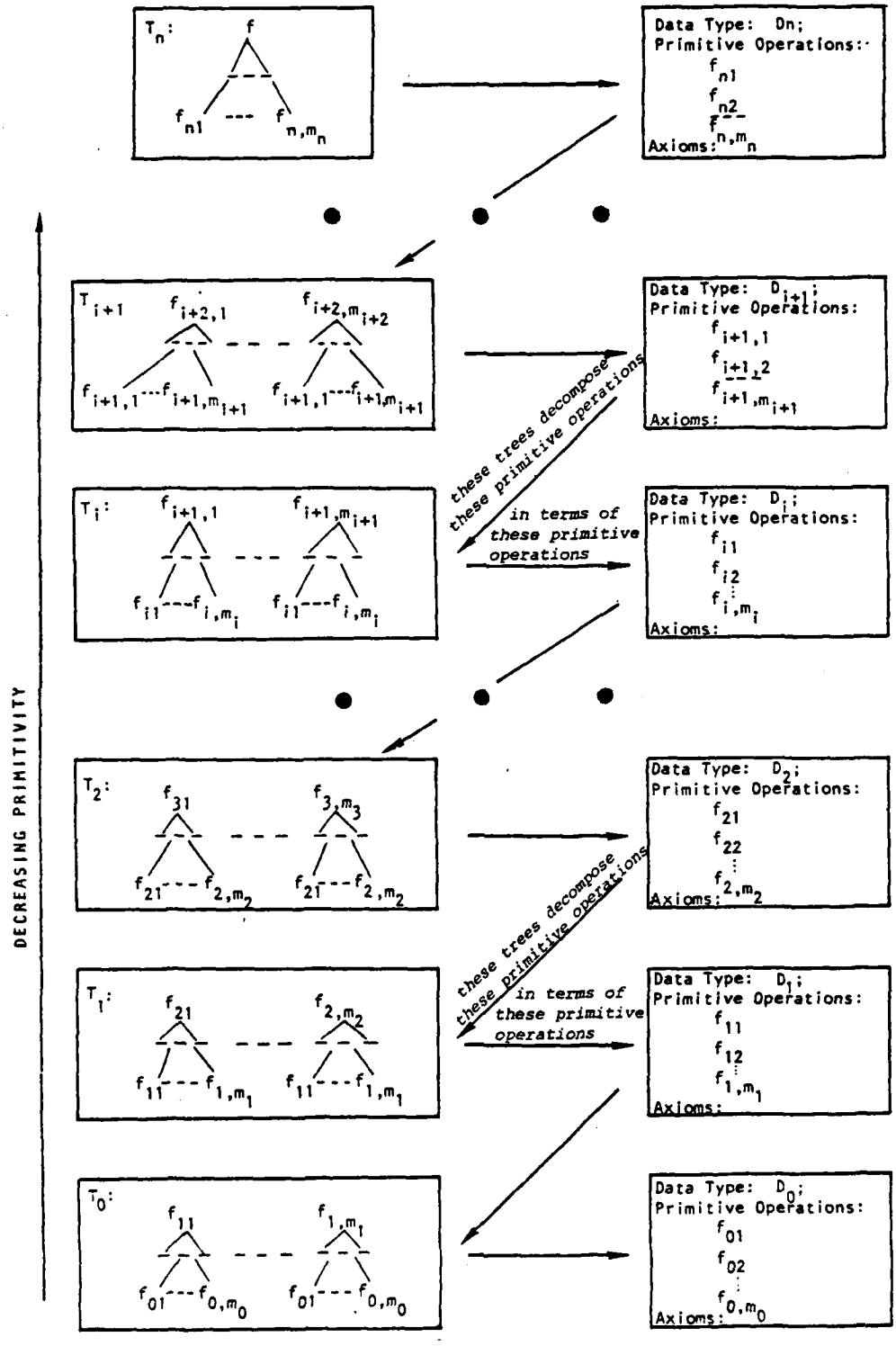


Figure 14

HOS Decomposition of Function f in terms of Primitive Operations of Data Type D_0

framework we have developed here and the SRI framework illustrated in Figure 5. The direction of the arrows in Figure 14 denotes flow of decomposition, however, rather than flow of implementation, as is the case in Figure 5. Everything in Figure 14 is strictly in the realm of specification and every subspecification ("module"), i.e., data types, trees, and data levels, is genuinely self-contained.

It is worth noticing at this point that Figure 14 suggests a way in which a relatively simple proof-of-correctness procedure might be developed for software specified in HOS. Robinson gives the following general account of how a proof-of-correctness procedure is supposed to work:

> The goal is to prove the correctness of a program P with respect to an input assertion, ϕ, and an output assertion, ψ. Verification requires the insertion of inductive assertions $\{q_i\}$ into the program's flowchart, breaking the program into simple paths. Each simple path has one entry and one exit and between these a fixed number of executable statements. For each simple path, a formula called a verification condition (VC) must be stated and proved to be a theorem. The validity of all the VCs for a program is sufficient to demonstrate the partial correctness of a program--i.e. for all inputs satisfying the input assertion, the output assertion is satisfied if the program terminates. Termination can be proved by inductive assertions (usually different from those used to prove partial correctness) that bound the number of loop executions... (p. 274).

If we view Robinson's description in terms of Figure 14, we get the following general picture. What we need in proof-of-correctness is a set of intermediate points in the specification of a function, at which correctness assertions (verification conditions) are stated and can be proven. In Figure 14, such intermediate points appear to be provided automatically at each data level, where the axioms on data types can be viewed as assertions on the decompositions of higher-data level primitive

operations into lower-data level primitive operations. The input assertions ϕ are provided by a statement of what, in general, we intended the specified function to do. Spelling out this procedure in detail will require further work, but the general idea would seem to be clear.

5. HIGHER ORDER SOFTWARE IS SECURE SOFTWARE

Now we are in a position to return to our main topic of security. Given the parallels that we have developed between the SRI "specification" methodology and HOS, it would undoubtedly be useful to examine the SRI security model in view of these parallels and see whether we can shed any light on how that model can be tightened up, as we did for Walter's. There is good reason for not doing this, however. The SRI notion of security is very similar to Walter's, as we can see from the following description of that notion by Feiertag:

> In a multilevel secure system there is a predefined set of security levels. The security levels are composed of clearances (or classifications) and category sets, but the composition of the security levels is an unimportant detail for purposes of this discussion and will be largely ignored[8]. What is important is that the security levels are partially ordered by the relation "less than" represented by "<". Each process in a multilevel secure system is assigned a security level. The processes may invoke functions that change the state of the system and return values. Each function instantiation (i.e., a function with a particular set of argument values) is assigned a security level. A process may only invoke those instantiations of functions that have been assigned the security level of the process. A system is multilevel secure if and only if the behavior of a process at some given security level can be affected only by processes at a security level less than or equal to the given level. Stated in terms of functions, this says that the values returned by a function instantiation assigned some security level can be affected only by the invocation of function instantiations at lower or equal security levels. Stated in loose terms this means that information can flow only upward in the system from processes of lower security level to processes of higher security level. [Fei76, p. 1].

We already have enough at our disposal, however, to solve the security problem altogether, without trying to reexamine Feiertag's model in light of HOS. Doing the latter can thus be left simply as an interesting exercise for the reader.

[8] Like Feiertag, Walter also informally characterizes a "classification" as consisting of a "sensitivity level" and a "compartment," but, also like Feiertag, this distinction plays no real role in his formal security model. Note that here, too, a secure model is characterized as one in which information can flow only upward.

We arrived at our HOS model in Figure 14 by sticking fairly closely to Robinson's SRI model, as illustrated in Figure 5, and showing that each component of his model could be made completely free of implementation by replacing it with the corresponding HOS notion. What we found, essentially, was that the step from implementation to specification can indeed be made in somewhat the way Robinson wants, but only if we reformulate his notions in non-implementation terms. To capture successfully what Robinson is trying to express, we have to replace his "abstract machines" with HOS data-type specifications and his "abstract programs" with HOS function-decomposition trees.

In fact, however, HOS is considerably more general than the model in Figure 14. In particular, there is no reason for the relationship between the primitive operations of successive data levels to be related as directly as Figure 14 suggests. In the figure, the primitive operations of one data level are decomposed directly into the primitive operations of the next lower data level. In general, however, there can be intermediate operations on the lower data level that mediate this decomposition.

As we noted earlier, a data level of HOS is an ordered pair (D,T), where D is a set of abstract data types and T is a set of decomposition trees. We also said the data levels are linearly (or partially) ordered and that they are related in that the decomposition trees of each data level decompose the primitive operations of the next higher data level in terms of the primitive operations of the lower data level. In the most general case, however, the decomposition trees on one data level also use the primitive operations of that data level at their terminal nodes to define operations that do not appear as primitive operations of the next higher data level. In this case, there will be further decomposition trees between the data levels whose roots are primitive operations of higher

data levels and whose leaves are primitive or non-primitive operations of next-lower data levels.

To put the point a little differently, a data level of HOS, from the most abstract point of view, is nothing more than an ordered pair (D,T), where D is a collection of sets and T is a collection of mappings (mathematical functions). What makes such an ordered pair an HOS data level, is the kinds of constraints that are imposed on D and T by the HOS axioms (and their consequences). Every member of D is not only a set, but a set whose members behave towards each other in a way specified in an HOS data-type specification. Every member of T is not only a mapping, but a mapping that is decomposed in a well-formed HOS tree.

The mappings in T can represent completely general functions and do not have to be primitive operations on either their own or any other data level. If a mapping f is non-primitive on its own data level, then there is a decomposition tree that connects it to the primitive operations of that data level. Such a tree can be said to be horizontal, because it relates primitive and non-primitive operations on a single data level. There is also, however, a vertical tree that relates f to the primitive operations of the next higher data level. In this tree f is one of the leaves and the root is one of the primitive operations of the higher data level.

What we get instead of the arrows in Figure 14, in other words, is a retroflex step structure like the one in Figure 15. Each line segment in Figure 15 represents a set of decomposition trees, some of which are horizontal (on a data level) and some of which are vertical (between two data levels). The arrows point away from the root nodes and toward the leaf nodes of the trees they represent. Filled circles represent primitive operations of a data level, while filled squares

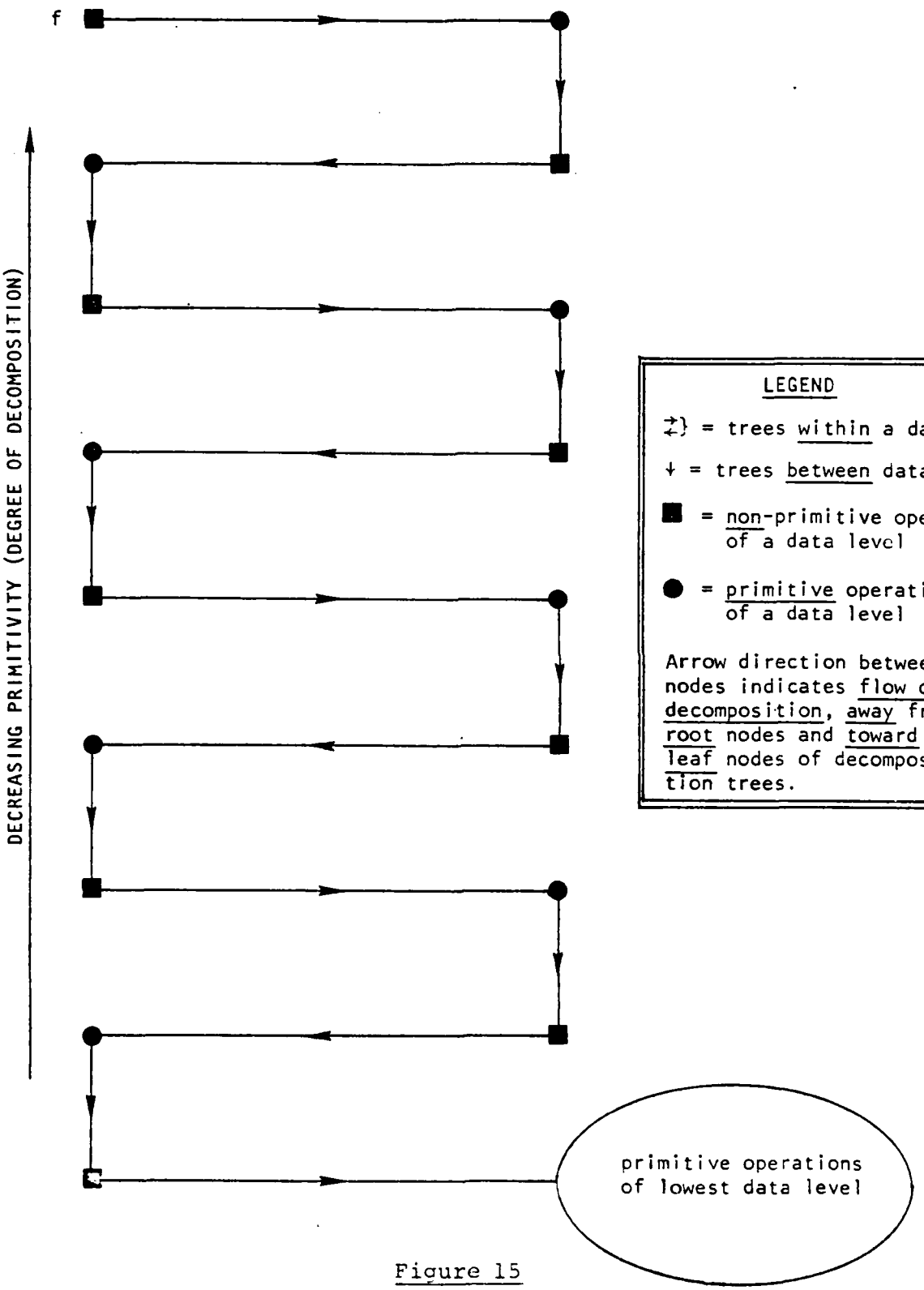


Figure 15

Retroflexed Step Structure of HOS Data Levels

denote non-primitive operations of a data level. Movement away from f produces increasingly decomposed operations (functions, mappings, etc.), i.e., an increasing degree of primitivity of the operations/functions involved. Movement towards f produces increasingly abstract or complex (decomposable) operations/functions, culminating in f itself.

In Figure 16, we elaborate this structure somewhat for a system with three data levels. As in Figure 15, filled circles in Figure 11 denote the primitive operations of a data level, while filled squares denote non-primitive operations of a data level. Open circles denote non-primitive operations that are needed in the intermediate levels of a decomposition tree[9]. These are described in terms of the three primitive control structures of HOS (composition, set partition, and class partition, as illustrated in Figure 13) or in terms of abstract control structures that are definable in terms of the three primitive control structures, as we mentioned in Section 4. Trees with solid branches are horizontal trees, which decompose non-primitive operations of a data level in terms of primitive operations of the same data level. Trees with dashed branches are vertical trees which decompose primitive operations of a data level in terms of non-primitive operations of the next lower data level. Note that primitivity of operations is a relative notion, defined with respect to the data level an operation is defined on.

Now we are ready to solve the security problem. Clearly, if we are not interested, for some reason, in the data-level structure of a particular f that has been decomposed as in Figures 15 and 16, then we can "fix" f in space, as it were, and "pull the rug out" from under the lowest data level, so that the filled nodes in the diagram act as pivots and the entire system stretches out into one gigantic tree structure, as in Figure 17. In conjunction with the HOS axioms, however,

[9] Note that HOS levels are defined relative to a controller, or parent module, whereas Robinson's are not. See [Ham76a,b,c].

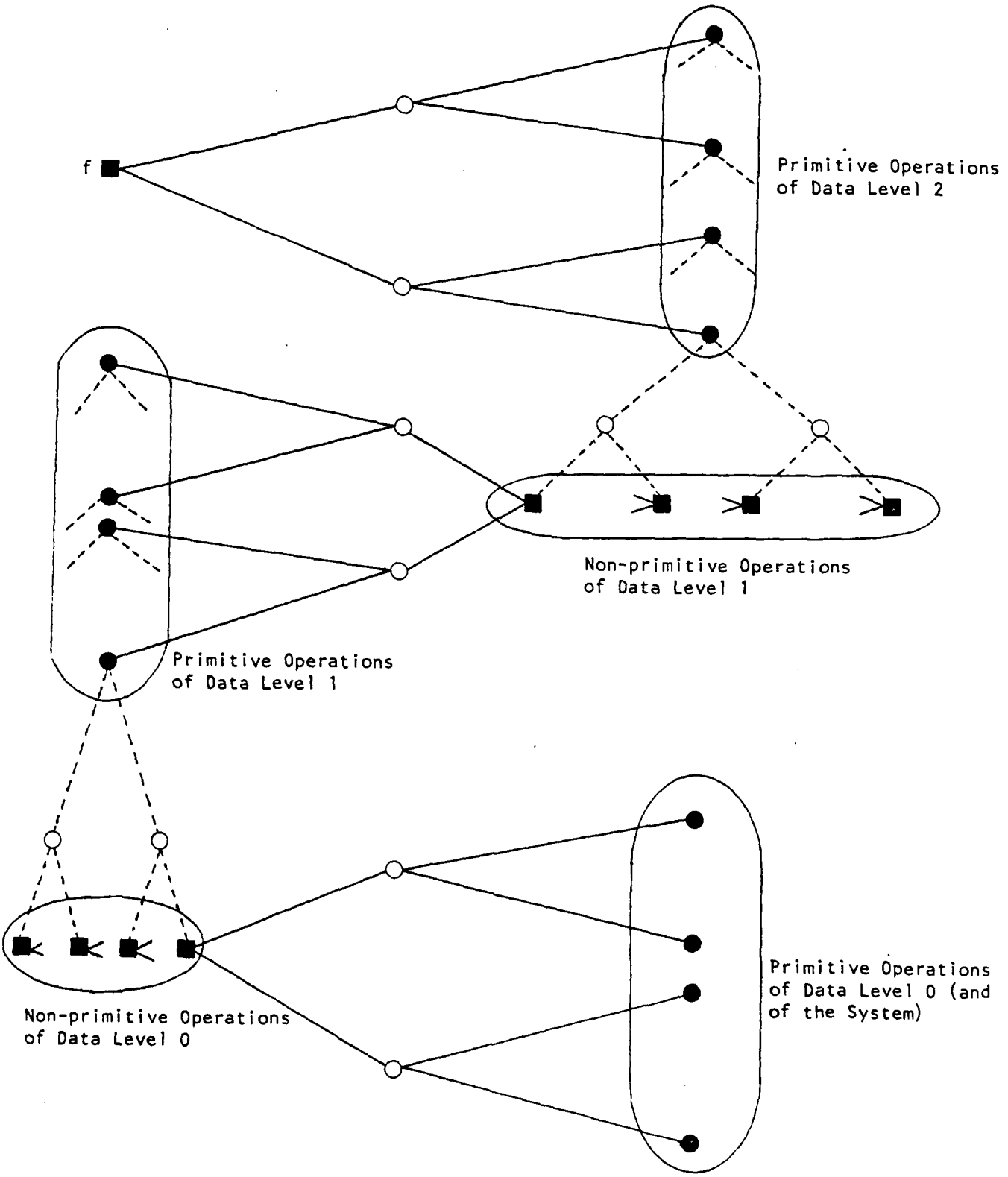


Figure 16

HOS Decomposition of Function f with Three Data Levels

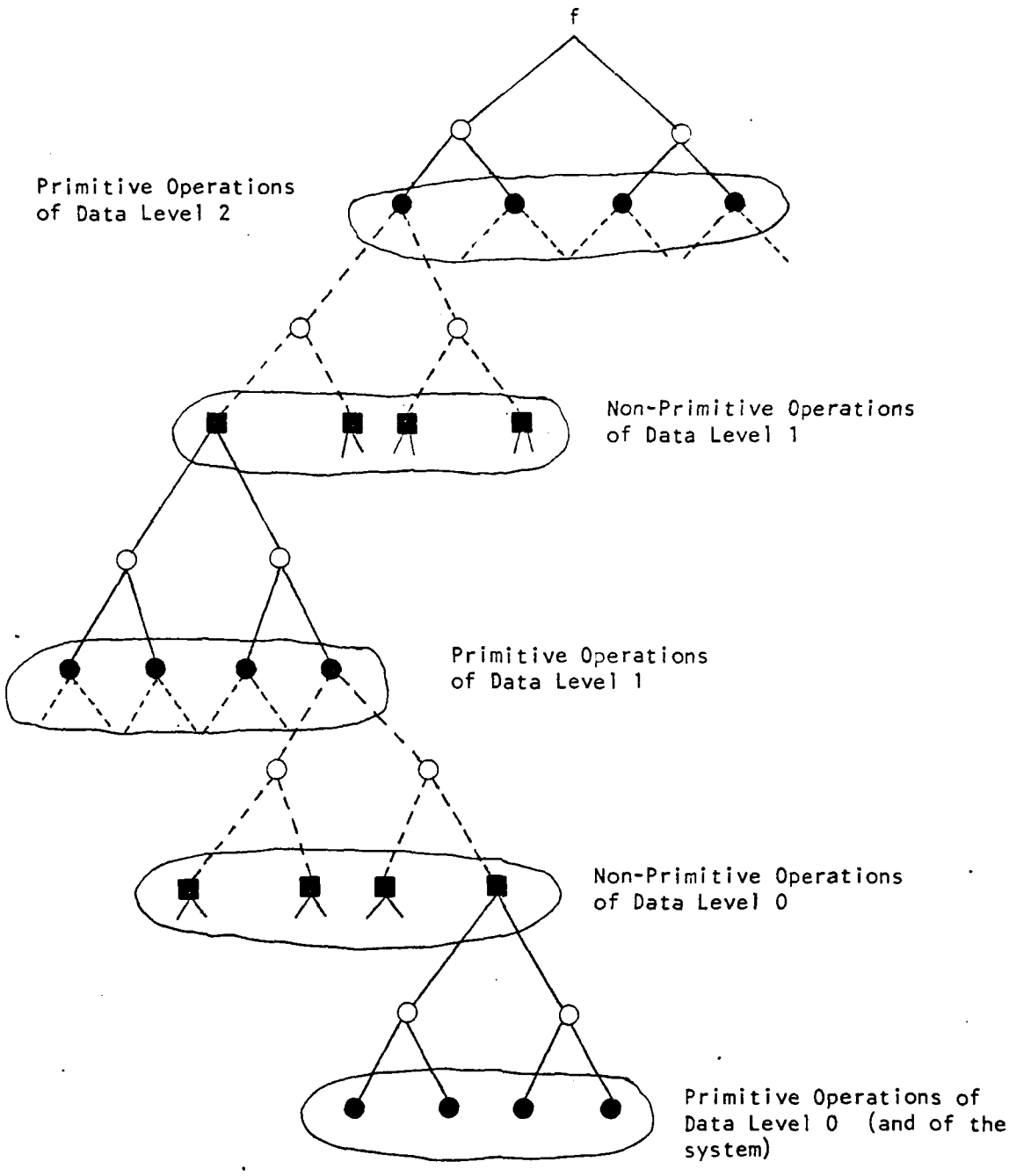


Figure 17

De-Retroflexed HOS Decomposition of Function f

this fact automatically provides us with the solution to the security problem.

Consider Axioms 3 and 4, in particular (Figure 12). These are the axioms that specify the access rights in an HOS system and would thus be expected to have something to do with security. Axiom 3 states that the access rights to the output of a function in a tree like that of Figure 17 are controlled by, and only by, its parent node ("module", in the axiom), i.e., the node immediately above it. Axiom 4, similarly, states that the access rights to the input of a function in such a tree is also controlled by and only by, its parent node. A given function can look at data only if its parent allows it to, and it must dispose of its results, again, only as its parent allows it to. It follows that the flow of control in an HOS system is always from the top down.

The flow of information, however, is always from the bottom up. A given node performs its function by having its offspring nodes, i.e., those on the immediately lower level, perform the function for it. This, in fact, is precisely what decomposition is really all about. Decomposing a function is just a formalized version of delegating responsibility. If someone can perform a task all by himself, then there is no point in delegating that task to subordinates. If responsibility is delegated, then he performs his task precisely by guaranteeing (via control) that his subordinates perform theirs. Formally, the offspring nodes look at the data that the parent allows them to (Axiom 4), perform their functions on that data as input, and then dispose of that data as the parent requires (Axiom 3), i.e., either by reporting it directly to the parent or by passing it on to an appropriate sibling. In particular, a given function has no idea what higher-level functions are doing. It just chugs along, turning input into output, disposing of that output as its parent tells it to. It is aware of what its offspring (or perhaps, siblings) are doing, however,

because that is precisely where it gets its input from in the first place.

The distinction between variables and values becomes very important here. Control is defined in terms of variables, but information is defined in terms of values. A node controls the access rights of its offspring to variables. The node tells the offspring what variables they can look at and what variables they must report back about. The offspring thus get their variables from the parent. This is the sense in which control flows downward. The parent node has no idea what the values of those variables are, however, until it gets those values from its offspring. The parent tells an offspring what variable to look at; then the offspring looks at the variable to find its value, operates on that value as input to change it into a value of an output variable, and then reports that value (either to a sibling or) back to the parent. Thus, while parents tell offspring what variables they can look at and assign, it is the offspring that tell the parents what the values of those variables are. It follows that information can flow only upward, precisely, in fact, because control flows downward, as stated in Axioms 3 and 4.

Our proof that information flows only upward in an HOS system required us to use the de-retroflexed tree in Figure 17, because the HOS axioms are stated in terms of trees (control maps), not in terms of retroflex trees, like the data-leveled structure in Figure 16. Since Figure 11 is functionally equivalent, however, to Figure 17, differing, in fact, only in its arrangement on the page, our proof of upward information flow is also valid for Figure 16.

The significance of this result cannot be overemphasized. As we saw in connection with Walter's M_0, a secure system is one in which the repositories (data) and agents (functions) are ordered in such a way, and the access rights of the agents

(functions) to the repositories (data) are assigned in such a way, that information can flow only upward in the ordering. What we have shown here, however, is that, if a system is specified in accordance with HOS, then its functions (agents) and data (repositories) are so ordered, and the access rights of the functions (agents) to the data (repositories) are so assigned, that information does always flow upward in the ordering. In other words, systems specified in HOS are automatically secure, without the need for any further paraphernalia to guarantee the security for us.

It follows that we have completely solved the security problem. If software is specified in HOS, then it is secure. It is that simple. Our proof of this fact also enables us to refine our discussion of HOS somewhat, and it would be worth while to pursue this opportunity a bit. We saw earlier that systems have a dual character in two distinct senses. On the one hand, a system is a function, since it performs a function, and it is also a datum in that it exists at all. On the other hand, a system consists of both data and functions and these two components are inseparable. What our proof of security makes clear, furthermore, is that each of these components has a dual character as well, and again, in two senses, when actually put together into a system.

A function in a system decomposition is a controller, because it controls the behavior of its offspring, in accordance with the axioms of HOS. It is also a performer, however, because it carries out the mapping of its parent. Every function plays both roles and the very fact that it plays one is the reason it must also play the other.[10] Data types also serve in two capacities in system decompositions. Every data type involved in a system decomposition provides both the input of one function and the output of another. "In" and "out" are as diametrically opposed as any two things can be, but,

[10] Primitive operations are also controllers (potentially), because we can always add a lower data level in which they are decomposed. Similarly, the highest-level function in a system is also a performer (potentially), because we can always use a system as a subsystem of some other system.

again, we cannot have one without the other.

Our components are also dual-natured in a second way. On the one hand, data have a constant aspect, as individual objects that can serve as inputs or outputs of functions, but, on the other hand, they have a variable aspect, because they exist as the members of data types. A given datum is an individual object itself, but it is also a representative member of a data type that can serve as a value of a variable of that type. This dichotomy enables functions to play a dual role in systems in a second sense as well. In Walter's terminology, a function can observe functions at a lower level of its decomposition tree by receiving data values from output variables of those functions and can modify functions at a higher level of its decomposition tree by providing data values to input variables of those functions.

On the one hand, therefore, functions act as agents, since they can observe lower-level functions and modify higher-level functions. What really gets observed and modified by these agents are the output variables of the respective functions with the modification occurring via the use of the input variables, so it is the output variables that serve as the repositories of the system. On the other hand, the input variables also function as agents because it is they that give the relevant values to their functions to use in modifying the values of the output variables. In general, in other words, it is the complete functions themselves--mappings, domains, and ranges, with the latter two represented by variables--that act both as agents and repositories in an HOS system.

It follows that we not only do not have to distinguish between repositories and security classes, as we saw earlier, but we do not really even have to distinguish between repositories and agents either. Since a function all by itself already

has a dual character, being made up of a mapping and two data types, functions themselves can play both roles. When functions occur in a tree, they can observe and modify other functions and they can also be observed and modified by other functions. Since they occur in a tree structure in any system, the functions themselves, and therefore the "agents" and "repositories," which the functions, are, are partially ordered, (and thus also pre-ordered), just as Walter wants them to be. A function in a system is both an agent and a repository and, since it occurs in a tree structure, can also serve as its own security class. This is about as cozy an arrangement as we could possibly want and, as we have seen quite clearly, it is absolutely secure.

6. SOFTWARE, SYSTEMS, SEMANTICS, AND BEYOND...

In most real scientific breakthroughs, the applicability, usefulness, and explanatory power of a new theory goes well beyond the restricted kind of problem it was originally intended to solve. All such developments clearly exhibit the contradictory aspects of similarity and difference, of continuity and change. Major breakthroughs always bear strong similarities to existing theories, but differ from them in key respects whose logical implications turn out to make all the difference.

These sorts of characteristic are clearly evident in the case of HOS as an approach to systems theory. We have seen that, while HOS was originally developed for the specification of reliable software, it turns out to provide automatically the solution to the security problem as well. Many HOS concepts look very much like the notions contained in other theories. Very close examination reveals, however, that HOS differs from other formulations in precisely the ways that are required by the problem the theories are trying to solve. What results is a completely adequate methodology for the specification of software systems that are reliable and secure.

In fact, what results is much more than that. HOS seems capable of providing insight into problems that are well beyond its intended field of software engineering. There is nothing in HOS, in other words, that requires us to restrict its use to specifying software systems. As a general systems theory, HOS can be fruitfully applied in any field in which systems can be seen to be playing a role. George Miller has suggested to us (personal communication) that HOS control hierarchies might be useful in accounting for the behavior induced by operant conditioning, and we have ourselves been investigating its usefulness as a tool

in analyzing natural language. The HOS distinction between data and functions, for example, can be interpreted as providing a semantic model, in the sense of Wilson [Wil76]., [Mill76]. Wilson's own semantic model, illustrated in Figure 18, is considerably less general, recognizing seven modes of existence, which he calls "concept types": objects, properties, relationships, events, actions, procedures, and sets. Such a model would certainly be useful for many purposes, but its limited generality cannot help but conceal significant generalizations that might help to simplify specific system specifications and suggest alternate implementations. Wilson's model represents a number of possible implementations of the HOS model at a lower level of generality. His "object classes," "property classes," "relations/attributes," and "sets," for example, could represent a particular selection of data types, with "events," "actions," and "procedures" corresponding to different classes of functions. The former, after all, represent things that are (be), while the latter represent things that do.

The data/function dichotomy could be distributed among Wilson's seven "concept types" in other ways as well, but the question that immediately strikes one on first coming across his model is that of why he chooses these seven in the first place. The problem with Wilson's semantic model, in other words, as a general systems (or semantic) theory, is that it is essentially ad hoc and, therefore, of limited generality. Actions certainly constitute events, for example, so they could be subsumed under them. Reversing direction, we could elaborate actions further, distinguishing them into transitive and intransitive actions, perhaps. Properties, similarly, could be taken to be one-place relationships, as they often are. The point is that Wilson's theory provides no natural mechanism with which to make the plethora of such decisions that might arise in specific cases of system design. The number of "concept types" is stipulated to be seven, in the theory itself, and that is that. HOS, in contrast, distinguishes only between

CONCEPT TYPES

Instances	Classes (Abstracted from Instances)
objects	object classes
properties	property classes
relationships	relations/attributes
events	event classes
actions	action classes
procedures	procedure classes
sets	set classes

CERTAIN KEY RELATIONSHIPS BETWEEN CONCEPTS

INSTANCE of/CLASS of

SUBCLASS of/SUPERCLASS of

COMPONENT of SUPERCOMPOSITE of (PART/WHOLE)

MEMBER of/SET MEMBERSHIP of

SUBSET of/SUPERSET of

Figure 18. Wilson's Semantic Model [Wil76, p. 7]

data and functions, while permitting any instance of either to be also an instance of the other. Given this generality, we can begin restricting things any way we like for any particular problem: three special data types and four classes of functions, two special data types and six classes of functions, ten special data types and one class of functions, etc. Once we let ourselves get more concrete than simply being versus doing, in other words, there is no clear general criterion for what our categories, modes of existence, or concept types should be, because each application or class of applications will favor a different choice. An adequate general systems theory must be formulated at the highest level of generality, so that no possibly desired choice of implementation will be ruled out, or made intrinsically more difficult, ahead of time.

By distinguishing only between data types and functions, in other words, HOS lets each particular more or less concrete application select exactly the specific data types and functions it needs, rather than arbitrarily stopping the theory short at a lower level of generality, and possibly ruling out the optimal choice of data types and functions for a particular application. The point here is not that Wilson's semantic model is wrong, but that, unlike HOS, it is not fully general, and, therefore, not fully adequate.

One final point remains to be made before we close. The reason that system specification has been such a difficult thing to figure out is that, as we have seen, a system is an intrinsically contradictory object. On the one hand, a system is a single object; on the other hand, it is made up of many different objects. On the one hand, a system performs a function on objects; on the other hand, it is an object on which functions can be performed. On the one hand, a system consists of two distinct kinds of objects, functions and data; on the other hand, functions can be data and data can be

functions, so only one sort of thing is really involved. A datum can be an input to a function, but it can be so only by being an output to a function, and vice versa. A function controls other functions, but it also gets controlled by another function. A datum is an individual object, with a constant aspect, and also a representative of a data type, with a variable aspect. Functions are defined in terms of variables, i.e., representatives of data types, but operate on constants, i.e., individual data, and so on.

Given all these twists and turns on the road to specification, it is not surprising that so many have lost their way. The uniqueness and power of HOS consists precisely in the fact that it manages to resolve all of these contradictions in one fell swoop and makes them comprehensible.

REFERENCES

[Cus77a] Cushing, S. "A note on equality in AXES data-type specifications" (in preparation).

[Cus77b] Cushing, S. and Heath, W. "ARO operating system," ARO Memo #1. Cambridge, MA: Higher Order Software, Inc. (hereafter cited as HOS, Inc.), April 7, 1977.

[Ham76a] Hamilton, M. and Zeldin, S. "AXES syntax description." TR-4. HOS, Inc., December 1976.

[Ham76b] Hamilton, M. and Zeldin, S. "Higher order software--a methodology for defining software." IEEE Transactions in Software Engineering, Vol. SE-2, No. 1, March 1976.

[Ham76c] Hamilton, M. and Zeldin, S. "Integrated software development system/higher order software conceptual description." Version 1. HOS, Inc., November 1976.

[Ham77] Hamilton, M. and Zeldin, S. "The manager as an abstract systems engineer." TR-5. HOS, Inc., June 1977. (To be presented at the COMPCON 77 Fall Conference, conducted by the IEEE Computer Society, Washington D.C., September 1977.

[Lind76] Linden, T. A. "Operating system structures to support security and reliable software." ACM Computing Surveys, VIII, 4. December 1976, pp. 409-445.

[Mill76] Mills, H. D. and Wilson, M. C. "An introduction to the information automat." Gaithersburg, MD: IBM, May 7, 1976.

[Par72a] Parnas, D. L. "A technique for software module specification with examples." Communications of the ACM, XV, 5. May 1972, pp. 330-336.

[Par72b] Parnas, D. L. "On the criteria to be used in decomposing systems into modules." Communications of the ACM, XV, 12. December 1972, pp. 1053-1058.

[Robi75] Robinson, L., et al. "On attaining reliable software for a secure operating system." Proceedings, International Conference on Reliable Software, Los Angeles. April 21-23, 1975.

[Robi77] Robinson, L., et al. "Proof techniques for hierarchically structured programs." Communications of the ACM, XX, 4. April 1977, pp. 271-283.

[Walt75] Walter, W. C., et al. "Structured specification of a security kernel." Proceedings, International Conference on Reliable Software, Los Angeles. April 21-23, 1975.

[Wil76] Wilson, M. L. "The information automat approach to design and implementation of computer-based systems." Gaithersburg, MD: IBM, April 1976.

Section III

SOME DATA TYPES FOR OPERATING SYSTEMS

by

S. Cushing and W. Heath

In "AXES Syntax Description*", we provided algebraic specifications for six intrinsic abstract data types: booleans, properties, sets, naturals, integers, and rationals. We have now developed algebraic specifications for a number of other abstract data types that are of particular usefulness in the specification of operating systems and other machine-dependent software. These include time, address, and two varieties of priority queue.

Time is an essential data type in any information processing system. It makes possible the scheduling of internal processes and provides an interface between the computer and external events. The essential characteristics of times are that they can be distinguished from each other (this is a universal characteristic of all data types, as far as we can tell); that they are linearly ordered, so that given two different times, one always precedes the other; and that they support a notion of time flow.

The data type TIME can be specified in AXES as in Figure 1. In the specification, Advance is the process of beginning at the time indicated by the first argument and advancing by the amount of time indicated by the second argument. Notafter is the relation that holds between two times if the first is earlier or simultaneous with the second. Equals is the relation of equality. Using Notafter, rather than something like Precedes, simplifies the axioms considerably. Precedes can be defined later as an operation.

Axioms 1-3 characterize equality simply as an equivalence relation, because the conceptual structure of the time notion is not rich enough to support a specific equality relation, such as the one on naturals or, as we will see, on queues. Axioms 4-6 impose a partial ordering on times and Axiom 7 makes the ordering total (linear). Axiom 8 characterizes Notime as the time for which Advance has no effect. Axiom 9 says that Advance is commutative, and Axiom 10 says it is associative. Axioms 8 and 9 together

*M. Hamilton and S. Zeldin, "AXES Syntax Description", TR-4. Higher Order Software, Inc., Cambridge, MA, Dec. 1976.

DATA TYPE: TIME;

PRIMITIVE OPERATIONS:

$time_3$ = Advance($time_1$,$time_2$);

boolean = Notafter($time_1$,$time_2$);

boolean = Equal($time_1$,$time_2$);

AXIOMS:

WHERE t,t_1,t_2,t_3 ARE TIMES;
WHERE Notime IS A CONSTANT TIME;

1. Equal(t,t) = True;
2. Equal(t_1,t_2) = Equal(t_2,t_1);
3. Entails(Equal(t_1,t_2) & Equal(t_2,t_3), Equal(t_1,t_3)) = True;
4. Notafter(t,t) = True;
5. Entails(Notafter(t_1,t_2) & Notafter(t_2,t_3), Notafter(t_1,t_3)) = True;
6. Entails(Notafter(t_1,t_2) & Notafter(t_2,t_1), Equal(t_1,t_2)) = True;
7. Notafter(t_1,t_2)! Notafter(t_2,t_1) = True;
8. Advance(t,Notime) = t;
9. Advance(t_1,t_2) = Advance(t_2,t_1);
10. Advance(t_1,Advance(t_2,t_3)) = Advance(Advance(t_1,t_2),t_3);
11. Notafter(Advance(t_1,t_2),t_1) = Notafter(t_2,Notime);

END TIME;

Figure 1

Data Type TIME

HIGHER ORDER SOFTWARE, INC. • 843 MASSACHUSETTS AVENUE • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 661-8900

guarantee that Notime behaves both as a time origin and as a null time increment. Axiom 11 relates Advance to the linear time ordering.

Addresses are necessary in computer-dependent software in order to keep track of storage locations. Addresses can be implemented as coordinates in a reference frame, as naturals indicating an order in a list of storage cells, as regions in a map, or as any number of other things, but all of these are irrelevant to the characterization of addresses as addresses. Addresses need not even be ordered, as long as we can tell which address is which. The regions of a map, for example, serve satisfactorily as addresses as long as we can tell which region on the map corresponds to which region in the mapped area, as in Figure 2:

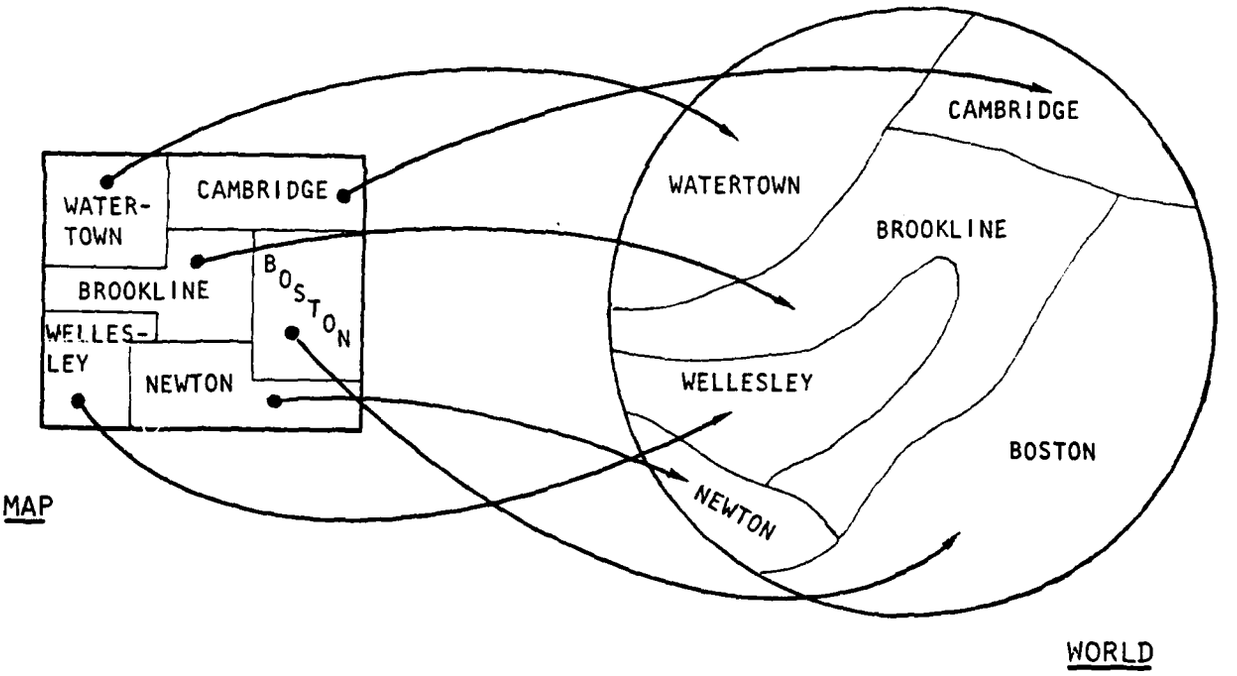


FIGURE 2

ADDRESSES IMPLEMENTED AS MAP REGIONS

Size and shape can be transformed beyond recognition, but the map regions still serve adequately as addresses as long as the transformation is topological and the correspondence is preserved.

The only requirement that needs to be made on addresses, in fact, is that we be able to tell them apart. This gives us the AXES specification in Figure 3. Since all we need is to be able to tell two addresses apart, equality is the only primitive operation. As with data type TIME, we characterize equality simply as an equivalence relation, because the conceptual structure of the ADDRESS type will not support a more specific relation.

Two forms of priority queue have been characterized algebraically and are now available for use in system specification. The differences between the two forms of queue, the strict priority queue and the flexible priority queue, are illustrated in Figure 4. Each of the queues in Figure 4 contains six items (presumably, jobs or processes), with corresponding items having equal priorities. Flow through both is from right to left, with the only exit point in each queue being at the left of the queue. The queues differ only in the points at which items may enter the queue.

In Figure 4a, entrance is strictly by priority; an item may enter the queue only at a point to the left of all items of lower priority and to the right of all items of the same or higher priority. Given a particular queue or state of a queue, a new item's entrance point is determined entirely by its own priority. No options are available.

In Figure 4b, entrance is bounded by priority, in the sense that an item may not enter the queue at a point to the left of any job of the same or higher priority; it may enter the queue at any point to the right of the rightmost item that has the same priority, if there is one, or the item of next highest priority, otherwise. Given a particular queue or state of a queue, a new item may enter anywhere to the right of the point determined by its priority.

HIGHER ORDER SOFTWARE, INC. · 843 MASSACHUSETTS AVENUE · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 661-8900

DATA TYPE: ADDRESS;

PRIMITIVE OPERATIONS:

$boolean_2$ = Equal($address_1$,$address_2$);

AXIOMS:

WHERE A_1,A_2 ARE ADDRESSES;

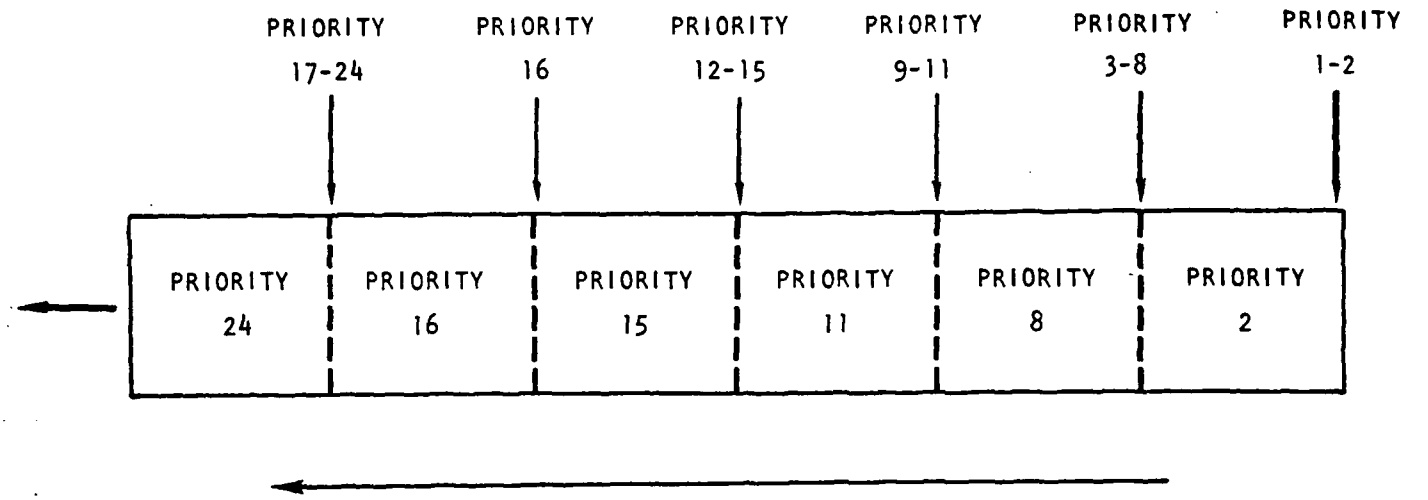
1. Equal(A_1,A_1) = True;
2. Equal(A_1,A_2) = Equal(A_2,A_1);
3. Entails(Equal(A_1,A_2) & Equal(A_2,A_3), Equal(A_1,A_3)) = True;

END ADDRESS;

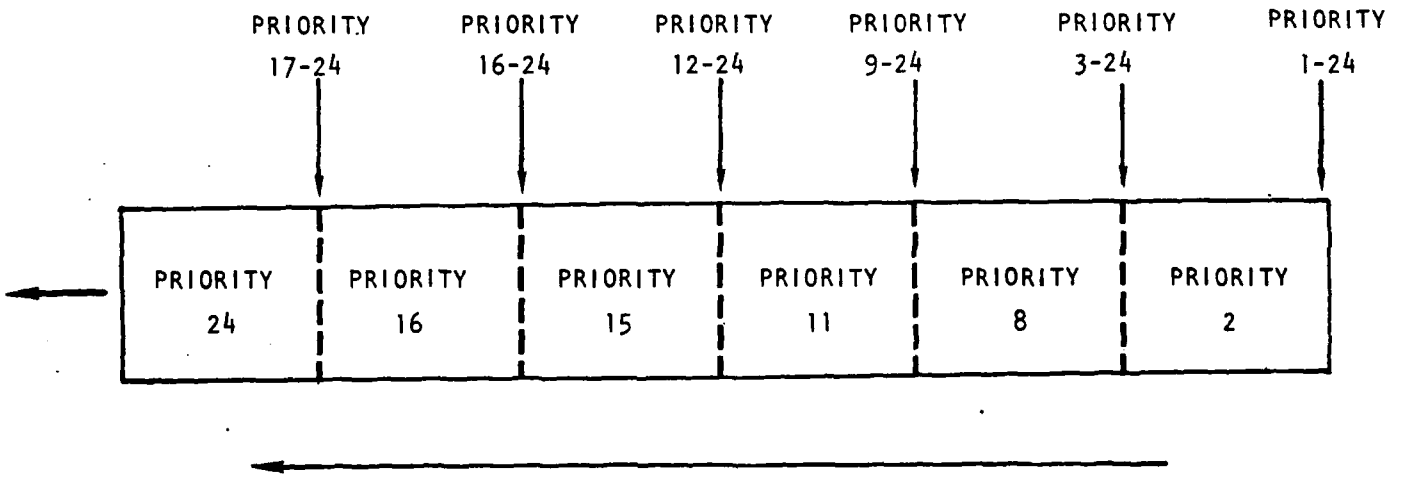
Figure 3

Data type ADDRESS

HIGHER ORDER SOFTWARE, INC. • 843 MASSACHUSETTS AVENUE • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 661-8900



(a) STRICT PRIORITY QUEUE: Entrance strictly by priority; arrows show exit point, entrance points, and direction of flow through queue.



(b) FLEXIBLE PRIORITY QUEUE: Entrance bounded by priority; arrows show exit point, entrance points; and direction of flow through queue.

FIGURE 4

TWO FORMS OF PRIORITY QUEUE

With a strict priority queue(Figure 4a), we automatically know where an item must enter, once we know its priority and the current state of the queue. With a flexible priority queue (Figure 4b), we know the leftmost point at which an item may enter, when we know its priority and the state of the queue, but we must then specify on some basis other than priority where to the right of that point we actually want it to enter.

It must be stressed that the "entrance points" in Figure 4 are abstract, in the sense that they might not actually appear in a particular implementation of data type PRIORITY QUEUE. A priority queue may be implemented in a form that actually has different entrance points, as in Figure 4, but it may equally well be implemented in a form that has only one entrance point at the right, as in a simple queue, with entrance being followed by a process that rearranges the queue in the appropriate way in accordance with the priority of the entering item. How the data type is implemented in an actual machine is irrelevant to the character of the data type as a data type. The important thing is that, once the queue-entering process, whatever that may involve, is completed for a particular entering item, the items are arranged in the queue correctly according to priority, along the lines of Figure 4.

Figure 5 contains an AXES specification of the flexible priority queue and Figure 6 contains an AXES specification of the strict priority queue. To make comparison easier, we have used the same notation for both. If both queues are actually used in a system specification, each should be provided with its own notation to avoid confusion. For example, Front could be called Flex-Front for the flexible priority queue, and SFront for the strict priority queue. Note that Priority is an operation defined on type JOB, which is not included in this memorandum.

The operation Add in these specifications is that of adding an item to a queue. Note that the two specifications differ only in that the Add operation for the flexible priority queue requires a natural number n as one of its arguments, whereas the

DATA TYPE: QUEUE;

PRIMITIVE OPERATIONS:

$queue_2$ = Add(natural,job,$queue_1$);

$queue_2$ = Remove($queue_1$);

job = Front(queue);

boolean = Equal($queue_1$,$queue_2$);

natural = Size(queue);

AXIOMS:

WHERE Nullq IS A CONSTANT QUEUE;
WHERE q,q_1,q_2 ARE QUEUES;
WHERE j,j_1,j_2 ARE JOBS;
WHERE Capacity IS A CONSTANT NATURAL;

1. Front(Nullq) = REJECT;
2. Front(Add(n,j,q)) = K_{REJECT}(^{1}(j,q)) AND ^{2}j AND Front(^{3}q);
3. Remove(Nullq) = REJECT;
4. Remove(Add(n,j,q)) = K_{REJECT}(^{1}(j,q)) AND ^{2}q
   AND Add(n,^{3}j,Remove(^{3}q));

PARTITION OF (j,q) IS
   ^{1}(j,q)|>(n,Priority(j))! Size(q)≥Capacity,
   ^{2}(j,q)|≤(n,Priority(j))&(Equal(q,Nullq).!>(Priority(j),
      Priority(Front(q)))) & Size(q)<Capacity,
   ^{3}(j,q)|≤(n,Priority(j))&Not(Equal(q,Nullq))&≤(Priority(j),
      Priority(Front(q)))& Size(q)<Capacity;

Figure 5
Data Type FLEXIBLE PRIORITY QUEUE

5. Equal(Nullq,Nullq) = True;

6. Equal(Nullq,Add(n,j,q)) = False;

7. Equal(Add(n,j,q),Nullq) = False;

8. Equal(Add(n_1,j_1,q_1), Add(n_2,j_2,q_2)) = Equal(n_1,n_2) & Equal(j_1,j_2) & Equal(q_1,q_2);

9. Size(Nullq) = Zero;

10. Size(Add(n,j,q)) = Succ(Size(^{3}q)) AND K_{REJECT}(^{4}q);

PARTITION OF q IS

^{3}q|<(Size(q), Capacity),

^{4}q|≥(Size(q), Capacity);

END QUEUE;

Figure 5

Data Type FLEXIBLE PRIORITY QUEUE (con't)

HIGHER ORDER SOFTWARE, INC. • 843 MASSACHUSETTS AVENUE • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 661-8900

DATA TYPE: QUEUE;

PRIMITIVE OPERATIONS;

$queue_2$ = Add(job,$queue_1$);

$queue_2$ = Remove($queue_1$);

job = Front(queue);

boolean = Equal($queue_1$,$queue_2$);

natural = Size(queue);

AXIOMS:
WHERE Nullq IS A CONSTANT QUEUE,

WHERE Capacity IS A CONSTANT NATURAL;

1. Front(Nullq) = REJECT;
2. Front(Add(j,q)) = ^{1}j AND Front(^{2}q) AND K_{REJECT}(^{3}q);
3. Remove(Nullq) = REJECT;
4. Remove(Add(j,q)) = ^{1}q AND Add(^{2}j,Remove(^{2}q)) AND K_{REJECT}(^{3}q);

PARTITION OF (j,q) IS

^{1}(j,q) | (Equal(q,Nullq) !> (Priority(j),Priority(Front(q)))) &
Size(q)<Capacity,

^{2}(j,q) | (Not(Equal(q,Nullq)) &≤ (Priority(j),Priority(Front(q))))
& Size(q)<Capacity,

^{3}(j,q) | Size(q)≥Capacity;

5. Equal(Nullq,Nullq) = True;
6. Equal(Nullq,Add(j,q)) = False;
7. Equal(Add(j,q),Nullq) = False;
8. Equal(Add(j_1,q_1), Add(j_2,q_2)) = Equal(j_1,j_2) & Equal(q_1.q_2);
9. Size(Nullq) = Zero;
10. Size(Add(j,q)) = Succ(Size(^{1}q)) AND Succ(Size(^{2}q)) AND K_{REJECT}(^{3}q);

END QUEUE;

Figure 6

Data Type STRICT PRIORITY QUEUE

strict priority queue does not require such an argument. The number n tells us where in the queue an item (job) is being inserted. In a flexible priority queue, n is bounded above by the priority of the entering item, but is not bounded below. In a strict priority queue, n is always identical to the priority of the entering item and so does not have to be stated.

The Remove operation removes the "leftmost" item (i.e., the front item, the one next in line to the exit point) from the queue. The Front operation tells us which item that is. Equal tells us when two priority queues are the same and Size tells us the number of items in a queue.

Axioms 1 and 2 characterize the interaction of Front, Add, and Nullq, the empty queue. Axiom 1 says that Nullq has no first item. Axiom 2 in Figure 5, along with the set partition, says that we cannot add an item to a queue at the position determined by priority n if n is greater than the item's own priority. Note that this condition is absent from Figure 6. In both figures, Axiom 2 says that if we add an item to a priority queue at an (the) appropriate position, then the new front item is the added item, if the old queue was empty or had a front item of lower priority than the added item, and is the old front item, otherwise.

Axioms 3 and 4 characterize the interaction of Remove, Front, and Nullq and are predictably similar to Axioms 1 and 2. Axiom 3 says that we cannot remove an item from the empty queue. Axiom 4 in Figure 5 says essentially the same thing as Axiom 2, if n is greater than the priority of the entering item. Again, this condition is absent from Figure 6. In both figures, Axiom 4 says that adding an item to a queue and then removing the front item from the new queue produces the old queue if the old queue was empty or had a front item of lower priority than the added item, and produces the same result as adding the new item to the old queue with its front item already removed, otherwise.

Axioms 5-8 characterize equality specifically for priority queues. Although the axioms for an equivalence relation follow from Axioms 5-8, the latter axioms narrow this relation down even further to equality. The conceptual structure of the notion of priority queue, unlike that of time or address, is rich enough to support such a characterization, as the axioms show. Axiom 5 says that Equality holds of Nullq and itself while Axioms 6 and 7 say that Equality fails to hold of Nullq and any queue that arises from the Add operation. Axiom 8 says that Equality holds of two queues that arise from Add, if and only if the added items were equal, were added to queues that were themselves equal, and were added at the same position in those queues.

Axioms 9-10 finish out our specification of priority queues by characterizing the notion of queue size. Axiom 9 says that the empty queue is of size zero, while Axiom 10 says that adding an item to a queue increases its size by one, unless the original queue was already filled to capacity, in which case the ADD operation REJECTs. Note that Axioms 2 and 4 tell us, through the Size-Capacity relationship in the partition, that both Front and Remove REJECT when applied after Add, if the original queue was too large to be added to in the first place, i.e., if it was already filled to capacity.

Section IV

SOME SPECIFICATIONS FOR THE OPERATING SYSTEM OF THE APOLLO GUIDANCE COMPUTER (AGC)

by

W. Heath

TABLE OF CONTENTS

HIGHER ORDER SOFTWARE, INC. • 843 MASSACHUSETTS AVENUE • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 661-8900

TABLE OF CONTENTS (continuation)

1.0 INTRODUCTION

What follows is the HOS specification of the operating system of the Apollo Guidance Computer (AGC). The AGC is a real-time control computer used to control the Apollo spacecraft.

The Apollo Guidance Computer is the heart of the Guidance, Navigation and Control System for the Apollo Lunar Module (LM) and Command Module (CM). The software maintains positional knowledge of the vehicle in space, determines the path to a desired destination, and steers the spacecraft along that path by sending commands to the engines. It communicates with the astronauts and the ground, and monitors the performance of the GN&C System. Mission programs, such as rendezvous, targeting and landing, control some of the phases of an Apollo flight.

Storage and Manipulation of Computer Instructions

The AGC contains two distinct memories, fixed and erasable, as well as various computer hardware. The fixed memory is stored in a wire braid which is manufactured and installed in the computer. This memory cannot be changed after manufacture and it can only be read by the computer. Fixed memory contains 36,864 "words" of memory grouped into 36 banks. Each word contains 15 bits of information (a sixteenth bit is used as a parity check). The word may contain either a piece of data, or an instruction which tells the computer to perform an operation. A series of instructions forms a routine or a program. In addition to storing programs, the fixed memory stores data such as constants and tables which will not change during a mission.

The erasable memory makes use of ferrite cores which can be both read and changed. It consists of 2,048 words divided into 8 banks. Erasable memory is used to store such data as may change up to or during a mission, and is also used for temporary storage by the programs operating in the computer.

Included in the hardware is a Central Processing Unit (CPU). The CPU performs all the actual manipulation of data, according to the instructions designated by a program. The 34 possible machine instructions include arithmetic operations (add, multiply, etc.) as well as logical operations, sequence control, and input/output operations. Also included are a limited number of "double-precision" instructions which permit two words of data to be processed as a single "word" of greater precision.

The memory cycle time (MCT) in the AGC is 11.7 μsec. Most single-precision machine instructions (e.g., addition) are completed in two MCTs; most double-precision machine instructions are completed in three MCTs. The unconditional transfer-control instructions, however, operate in one MCT.

To be used as an instruction, a computer word must specify the operation to be performed and give the location of the data to be operated on. However, a 15-bit word does not contain enough information to specify 34 operations and 38,912 fixed and erasable locations. In fact, 15 bits cannot even specify 38,912 locations unambiguously. It is for this reason that both the fixed and the erasable memories are grouped into banks. An instruction may specify any address within its own bank, and may also address the first four banks of erasable and the first two banks of fixed memory. Access to other banks is accomplished using bank-selection registers in the CPU. In many cases a program exists entirely within one bank of memory, in which case bank switching is not required.

Many of the tasks the AGC performs can be adequately carried out by machine instructions. However, for extensive mathematical calculations in such areas as navigation,the short word length of the AGC presents difficulties. It limits the number of instructions available, the range of memory that can be addressed without switching banks, and the precision with which arithmetic data can be stored and manipulated. To alleviate these problems, nontime-critical mathematical calculations are coded in "interpretive language" and are processed by a software system known as Interpreter. Each Interpreter instruction is contained in two or more consecutive computer words. The increased information available allows more possible instructions and a greater range of memory addressability without bank switching. In fact, with some exceptions, all of erasable memory and fixed memory may be addressed directly. Among the available Interpreter instructions are a full set of operations on double-precision quantities, including square root and trigonometric functions, some triple-precision instructions, and a set of vector instructions such as cross product, dot product, matrix multiply, and vector magnitude. Interpreter routines translate an Interpreter instruction into an equivalent series of machine instructions to be performed by the CPU. Thus, one Interpreter instruction may be equivalent to many machine instructions, and much storage space is saved in the computer. The Interpreter also contains software routines for the manipulation and temporary storage of double- and triple-precision quantities and vectors.

Interpreter expands the processing capabilities of the CPU hardware. However, its operation is quite slow, since the CPU must perform all the actual operations, and much time is spent in the translation of instructions and the manipulation of data. Although processing time is slower, much storage space is saved in fixed memory by the more powerful Interpreter instructions; thus, the vast majority of nontime-critical mathematical computations are coded using interpretive language.

Timing and Control of the Computer

Two of the more stringent requirements placed upon the AGC are the need for real-time operations and the necessity for time-sharing of multiple tasks.

Certain computer functions must occur in real time. For example, certain input data must be stored or processed immediately upon receipt; and outputs, such as those which turn the jets on and off, must occur at precisely the correct time. An interrupt system causes normal computer operation to be suspended while performing such time-critical tasks.

Several programs, which are less time-critical, may all be required during a phase of the mission. Time sharing between these programs is controlled by a software executive system which monitors the programs and processes them in order of priority. The Executive can stop one job when a higher priority job is necessary, then resume the low-priority job when time is available.

Interrupt System

To permit quick response to time-dependent requests, the AGC has a complex interrupt structure. There are two classes of interrupts, counter interrupts and program interrupts. Counter interrupts have the highest priority of all AGC operations. Counters are locations in erasable memory which can be modified by inputs originating outside the CPU. Some counters are used as clocks, while others interface with spacecraft systems to receive or transmit sequences of data pulses. The counters respond to a set of involuntary instructions called counter interrupts, which may increment, decrement, or shift the contents of the counters. A counter interrupt suspends the normal operation of the CPU for one MCT, while the instruction is being processed. Except for the short time loss, the ongoing program is not affected by the counter interrupt; in fact, it is not aware that the interrupt has occurred. These interrupts are used solely for counter update and maintenance; their priority assures that no information will be lost in the counters.

Two counters, designated TIME1 and TIME2, form a double-precision master clock in the AGC. TIME1 is incremented at the rate of 100 counter interrupts per second. Overflow of TIME1 triggers a counter interrupt to increment TIME2. Since total time that must elapse before TIME2 overflows exceeds 31 days, TIME1 and TIME2 are thus able to keep track of total elapsed mission time.

The remaining clock-counters, designated TIME3 through TIME6, measure time intervals needed by the AGC hardware and software. For example, autopilot computations must be processed periodically whenever the autopilot is in use. Before reaching completion, these computations preset the TIME5 counter so that it will overflow at a specified time in the future. When TIME5 overflows, a signal sent to the CPU causes a "program interrupt" which interrupts the program in process and begins the autopilot computations once again.

Program interrupts have lower priority than counter interrupts, but greater priority than normal program operation. Unlike counter interrupts, the purpose of program interrupts is to alter the normal processing sequence. There are 11 program interrupts; they may be triggered by a clock-counter overflow, as in the example given above, or by externally generated signals, such as the depression of a key on the Display and Keyborad (DSKY) by an astronaut. The occurrence of a program interrupt causes the computer to suspend normal operation at the end of the current instruction. The current CPU data are saved, the computer is placed in interrupt mode, and control is passed to a preassigned location in fixed memory. This preassigned location is the beginning of a program which performs the action appropriate to the interrupt. While the interrupt program is running, the computer remains in interrupt mode, and no additional program interrupts will be accepted, although counter interrupts can still occur. (Requests for other program interrupts are stored by the hardware and processed before returning to normal operations.) At the conclusion of the interrupt program, a "resume" instruction is executed. If there are no other program interrupts, the CPU is taken out of interrupt mode, the original contents are restored, and the program returns to the point at which it was interrupted. One program interrupt (restart) takes precedence over all the others, and can even interrupt an interrupt. It results from various kinds of computer malfunctions.

A computation which takes place by means of a program interrupt is called a task. Since tasks may not be interrupted, they must be short to avoid delaying other tasks. This speed requirement precludes the use of interpretive language.

A job in process must periodically call Executive to scan the list of waiting jobs, thus determining if any scheduled job has a priority higher than itself. If so, the job currently active is suspended and the higher priority job is initiated. To permit suspension of a job and subsequent resumption at a point other than its beginning, the working storage associated with the job is saved when the job is suspended and restored when the job is reinstated. A suspended job is returned to the job list and is not reinstated until it has the highest priority on the list. Eventually, a given job will run to completion, at which time it is removed entirely from consideration. When all jobs on the list have run to completion, a "DUMMYJOB" with zero priority constantly checks to see if new jobs have appeared. (The computer also performs a self-check.)

The relative importance of a job may change for various reasons. When this is the case, Executive changes the priority list and rechecks the list for the job of highest priority. Many times it is desirable to purposely suspend the execution of a job, but not to terminate it completely. Temporary suspension is desirable to await an event such as the input or output of data, or for the availability of a nonreenterable subroutine currently in use. To accomplish temporary suspension, Executive saves the job's interrupted registers and sets its priority to a negative value. Because the interrupted job has a negative priority, DUMMYJOB has priority over it. As a result, the job is, in effect, suspended indefinitely. Eventually, Executive is called to restore the job, usually by the event for which the job is waiting. Executive restores the original priority and again checks the list for the highest priority job. (This process is called "putting a job to sleep.")

Waitlist allows any program to schedule a task to occur at a specified time in the future. The TIME3 clock interrupts the job in process at the correct time and initiates the task. (As mentioned before, tasks initiated by the other program interrupts are not controlled by the Executive.)

To schedule a new task, Waitlist requires the starting address of the task and the amount of time which must elapse before execution. Waitlist maintains a list of tasks waiting to run in the order in which they will be performed and a list of time differences between adjacent items on the task list. It determines when the new task will run in relation to others on the list, placing it appropriately in the list.

One class of tasks is initiated by overflow of the time counters TIME3, TIME4, TIME5, and TIME6. These are considered time-dependent tasks. The TIME5 interrupt, described described above, initiates autopilot computations at precise periodic intervals. TIME6 controls the timing of the autopilot RCS (Reaction Control System) jet firings. TIME4 initiates a series of routines which periodically monitor the IMU, radar, etc., and process input/output commands. The TIME3 counter is under the control of the software executive system (described below). It is available for general use by any program needing to schedule a task for a specific time.

A second class of tasks is initiated by interrupts caused by external action. For example, depressing a DSKY key initiates a task that begins processing DSKY readings and storing the information for later processing. Telemetry and the radar also cause interrupts that initiate tasks to receive or transmit the next data word.

Software Executive System

Computation in the AGC is managed by a software executive system comprised of two groups of routines, Executive and Waitlist. This system controls two distinct types of computational units, jobs and tasks. In its normal operating mode, the computer processes jobs. These are scheduled by the Executive, according to a priority system. The Waitlist uses the TIME3 interrupt to schedule tasks for a specific time in the future. (Tasks originated by the other program interrupts take place independently of the software executive system.)

Most AGC computations are processed jobs. Division of a program into discrete jobs is at the discretion of the programmer, who also assigns a priority to each job indicative of its importance. The Executive can manage up to seven jobs (eight in the LM program) simultaneously.

To schedule a job, the Executive places the job's priority and beginning location on a list, assigning the job a set of working storage locations called a core set. In addition, if a job requires a larger working storage, as in the use of interpretive language, a second area, called a VAC area, may be assigned. The Executive is capable of maintaining seven core sets (eight in the LM program) and five VAC areas as each is assigned to a job, and of redesignating them as available when the job is finished.

The TIME3 counter counts the time to the first item on the list. When this time arrives, the TIME3 program interrupt occurs. TIME3 is immediately set to overflow when the time has elapsed for the next task on the list, and all tasks and times move up one position on the list. The computer remains in interrupt mode until the task is completed. It is then free to process other interrupts or return to the original job.

Since TIME3 is a single precision AGC word (15 bits) that is incremented 100 times a second, Waitlist can process tasks up to 162.5 sec in the future. For longer delays, a routine called LONGCALL processes a single task, the repeated calling of Waitlist. LONGCALL can schedule tasks for as long as 745 hours in the future, a time span larger than an entire Apollo mission.

Sequence Control

In normal AGC operation, the Executive maintains a constant background of activity, while program interrupts break in for short, time-critical bursts. The execution of a job is subject to numerous interruptions. A counter interrupt may occur after the completion of any instruction. Program interrupts stop the job in process. While the computer is in interrupt mode, any further program interrupts are saved by the hardware and processed one at a time before returning to the job. Under control of the Executive, high-priority jobs also steal time from a job in process. This control system of interrupts and priorities ensures that in times of heavy load, the most critical computations for the mission will be processed first.

Normally, the CPU does not stop during periods of low activity. If no jobs or tasks are being executed, the CPU executes a short loop of instructions (DUMMYJOB) which continually looks for jobs to initiate. Periodically, TIME4 overfolows, initiating a task to monitor various GN&C subsystems. If an autopilot is in operation, TIME5 triggers other interrupts for autopilot functions. In addition, periodic counter interrupts will occur as counter input is received and clock counters are updated. More extensive computer activity awaits action by the astronaut.*

---

*Abstracted from Johnson, M.S. and Giller, D.R. "MIT's Role in Project Apollo", The Software Effort, vol. 5. Draper Laboratory, Inc., Report R-700, Final Draft, March 1971.

The HOS specification of the AGC operating system has been partitioned into two sections. The first is the Waitlist System which controls the execution of tasks. A task, as in the actual AGC software, is a short process which runs to completion once started and is scheduled according to a specified time of exectution. The second section of specifications is the Executive System which controls the execution of jobs. A job, again identical to the AGC definition, is a process of arbitrary length which is established as a job when its function is desired and is subsequently scheduled according to a priority that is specified for the process, and is subject to interruptions for higher priority jobs. Section 2 contains the HOS specification ot the Waitlist System. Section 3 contains the Executive System specifications.

A comment on notation: The symbols "&" and "!" have been used herein to denote the boolean infix operators "AND" and "OR", respectively.

2.0 WAITLIST SYSTEM SPECIFICATION

The Waitlist System specification consists of a set of data type specifications and a set of operation specifications. Section 2.1 contains the data types and section 2.2 contains the operations. The use of the data types and service operations (those operations accessible by the user) is outlined below.

Waitlist System Data Types:

- TASK_QUEUE: A queue of requests for task invocations. Task invocations are ordered in the queue according to their requested execution times. The queue has a finite size and is updated by the Waitlist System operations.

- TASK_ENTRY: A storage device to perserve, for entry into a TASK_QUEUE, a task identifier and the time delay for the execution of the task.

Waitlist System Service Operations:

- Enter_Task: A primitive operation on the data type TASK_QUEUE. Enters a task invocation request into the queue for execution after its specified time delay. (Corresponds to the AGC program entries: WAITLIST and TWIDDLE, where TWIDDLE is a different implementation using a shorter form of address reference.)

Waitlist System Service Operations (continued):

- Longcall: Invoked to request the execution of a task with a greater time delay than the maximum allowable by the TASK_QUEUE. (Corresponds to the AGC program entry: LONGCALL.)

- Taskover: Must be invoked at the completion of each task to determine if another task is waiting for immediate execution. (Corresponds to the AGC program entry: TASKOVER.)

- Counter_Interrupt: Invoked to synchronize the Waitlist System with real time. The AGC erasable register TIME3 is used to keep track of the time delay until the next required task execution. This information has been incorporated into the TASK_QUEUE data type as the Delta_T value of the first TASK_ENTRY in the queue. In the specification, this operation is invoked to decrement and test this time delay value. (Corresponds to a hardware counter interrupt in the AGC.)

- T3rupt: Invoked to service the TASK_QUEUE and present the task for execution when Counter_Interrupt indicates that a task is ready to by executed. (Corresponds to the AGC program entry: T3RUPT, which is a task itself that is executed in the program interrupt caused by the overflow of TIME3.)

2.1 Waitlist System Data Types

The data type specifications of the Waitlist System are contained herein. Each specification contains the AXES syntax for the specification and appropriate explanations of the primitive operations and the axioms of the data type.

DATA TYPE: TASK_QUEUE(n);

PRIMITIVE OPERATIONS:

task_queue$_2$ = Enter_Task(task_queue$_1$,task_entry);

task_queue$_2$ = Discard_Task(task_queue$_1$);

task_entry = First_Task(task_queue);

natural = Size(task_queue);

natural = Capacity(task_queue);

boolean = Equal(task_queue$_1$,task_queue$_2$);

AXIOMS:

WHERE n,n_1,n_2 ARE NATURALS;

WHERE EmtyTskQ$_n$ IS A CONSTANT TASK_QUEUE(n);

WHERE e_1,e,se,ne ARE TASK_ENTRYS;

WHERE q_1,q_2,q_3,q,sq,nq,qi,sqi,nqi ARE TASK_QUEUES;

WHERE DummyTask IS A CONSTANT TASK_ENTRY;

/*1*/ First_Task(EmtyTskQ$_n$) = DummyTask;

/*2*/ Discard_Task(EmtyTskQ$_n$) = EmtyTskQ$_n$;

/*3*/ Size(EmtyTskQ$_n$) = 0;

/*4*/ Capacity(EmtyTskQ$_n$) = n;

/*5*/ Equal(EmtyTskQ$_{n_1}$,EmtyTskQ$_{n_2}$) = Equal(n_1,n_2);

/*6*/ Equal(q_2,q_3) = Equal(q_3,q_2);

DATA TYPE: TASK_QUEUE(n); (continued)

/*7*/ Enter_Task(sqi,se) = WHEREBY q OTHERWISE K_{REJECT}(nqi,ne,nq);

PARTITION OF (sqi,se,sq) IS

(qi,e,q) | Size(sqi) < Capacity(sqi),

(nqi,ne,nq) | Size(sqi) $\geq$ Capacity(sqi);

/*8*/ q $\neq$ $EmtyTskQ_n$;

/*9*/ Size(q) = 1 + Size(qi);

/*10*/ Equal(q,Enter_Task(q_1,e_1)) = Equal(qi,q_1) & Equal(e,e_1);

/*11*/ First_Task(q) = ^{1}e OTHERWISE First_Task(^{2}qi);

/*12*/ Discard_Task(q) = ^{1}qi OTHERWISE Enter_Task(Discard_Task(^{2}qi),^{2}e);

PARTITION OF (qi,q,e) IS

^{1}(qi,q,e) | Preceeds?(Delta_T(e),Delta_T(First_Task(qi))),

^{2}(qi,q,e) | -Preceeds?(Delta_T(e),Delta_T(First_Task(qi)));

/*13*/ Capacity(q) = Capacity(qi);

END TASK_QUEUE(n);

DATA TYPE: TASK_QUEUE(n); (continued)

Primitive Operation Descriptions:

q_2 = Enter_Task(q_1,e);	Enters a TASK_ENTRY into a TASK_QUEUE according to its Delta_T. An entry is queued behind all others of lower Delta_T and all others of the same Delta_T that were filed before it. It is queued before all entries of higher Delta_T and all entries of the same Delta_T that are filed after it. Implictly, once a task is entered into the queue, its Delta_T becomes the time-difference between its execution time and the execution time of the task preceeding it in the queue. Thus, if two tasks in the queue have the same execution time, the Delta_T of the second would be equal to Notime. In this way, when a task becomes first in the TASK_QUEUE, its Delta_T represents the time delay between the current time and the task execution time.
q_2 = Discard_Task(q_1);	Eliminates the first entry in the TASK_QUEUE. Will never reject (see Axiom 2).
e = First_Task(q);	The first entry in the TASK_QUEUE. There is always a first entry because DummyTask is always returned as the first task of an EmtyTskQ (see Axiom 1).
n = Size(q);	The current size of the TASK QUEUE (i.e., number of entries present). A DummyTask is not included in this count (see Axioms 3 & 9).

DATA TYPE: TASK_QUEUE; (continued)

n = Capacity(q);	The maximum allowable size of the TASK_QUEUE.
b = Equal(q_1,q_2);	Test to determine if two TASK_QUEUES are identical.

Axiom Descriptions:

Axiom 1: Assures that DummyTask is always present in an EmtyTskQ.

Axiom 2: Assures that DummyTask will be present as the first entry in an EmtyTskQ even after an entry is discarded.

Axiom 3: In conjunction with Axiom 9, states that Size is a count of the entries in the queue that are not DummyTask

Axiom 4: The capacity of a TASK_QUEUE is determined by the subscript of the $EmtyTskQ_n$ that it was built from.

Axiom 5: Two $EmtyTskQ_n$'s are identical if their capacities are equal (as indicated by their subscripts).

Axiom 6: States that the test for equality is reflexive.

Axiom 7: States that Enter_Task will reject if the queue is already filled to capacity. In this statement, also, is defined a symbol for the case in which the queue is not previously filled to capacity.

Axiom 8: Indicates that an EmtyTskQ is never the same as a TASK_QUEUE in which an entry has been made.

DATA TYPE: TASK_QUEUE(n); (continued)

Axiom 9: States that an entry made into a queue that is not filled to capacity will increase its size by one. Note that if DummyTask is entered via the Enter_Task operation, it will be treated as any other task and will increase the size of the queue.

Axiom 10: States that the TASK_QUEUES resulting from entries made into two non-filled queues will be identical if the original queues were identical and the two task entries are identical.

Axioms 11 & 12: Assure that the process of entering a task into the queue will:

1. place the new task at the front of the queue if it has the smallest Delta_T, or
2. place the new task in its proper order so that repetitive applications of Discard_Task and First_Task will find the new task in front of all tasks with greater Delta_T.

Axiom 13: The capacity of the queue is invariant over the Enter_Task operation.

```
DATA TYPE:  TASK_ENTRY;

PRIMITIVE OPERATIONS:

     address      = Task_Addr( task_entry );

     time         = Delta_T( task_entry );

     task_entry2  = Asgn_Task( task_entry1,address,time );

     boolean      = Equal( task_entry1,task_entry2 );

AXIOMS:

     WHERE  EmtyTask  IS A CONSTANT TASK_ENTRY;

     WHERE  e,e1,e2  ARE TASK_ENTRYS;

     WHERE  a  IS AN ADDRESS;

     WHERE  t  IS A TIME;

/*1*/     Task_Addr( EmtyTask ) = REJECT:

/*2*/     Delta_T( EmtyTask ) = REJECT;

/*3*/     Task_Addr( Asgn_Task(e,a,t) ) = a;

/*4*/     Delta_T( Asgn_Task(e,a,t) ) = t;

/*5*/     Equal( e1,e2 ) = Equal( Task_Addr(e1),Task_Addr(e2) )
                           &  Equal( Delta_T(e1),Delta_T(e2) );

END TASK_ENTRY;
```

DATA TYPE: TASK_ENTRY; (continued)

Primitive Operation Descriptions:

a = Task_Addr(e);	ADDRESS attribute of a TASK_ENTRY.
t = Delta_T(e);	TIME attribute of a TASK_ENTRY.
e_2 = Asgn_Task(e_1,a,t);	Assigns the ADDRESS and TIME attributes to a TASK_ENTRY.
b = Equal(e_1,e_2);	Tests to determine if two TASK_ENTRYs are identical.

Axiom Descriptions:

Axiom 1: An EmtyTask has no ADDRESS attribute to examine.

Axiom 2: An EmtyTask has no TIME attribute to examine.

Axiom 3: The ADDRESS attribute that is assigned to a TASK_ENTRY becomes the value that may be examined.

Axiom 4: The TIME attribute that is assigned to a TASK_ENTRY becomes the value that may be examined.

Axiom 5: Two TASK_ENTRYs are identical if their assigned attributes are identical.

A TASK_ENTRY consists of the task address and delay time for an entry into a TASK_QUEUE. The task specified by the address will be executed after a delay specified by the time when an entry is made into the queue. EmtyTask is the empty value of a TASK_ENTRY.

2.2 Waitlist System Operations

The specifications of all the Waitlist System operations are contained in this section. A description of the function of the operations is given below. The specifications follow.

Waitlist Operation Descriptions:

Preceeds?:	Test to determine if one time occurs before another.
Reverse:	Produces a value that is the time reverse of the input.
Regress:	Decrements one time value from another.
New_Task_Entry:	Generates a TASK_ENTRY from the specified attributes.
Update_Waittime:	Replaces the Delta_T of the first TASK_ENTRY in the TASK_QUEUE by the new time delay value that is specified.
Pop_First_Task:	Removes the first TASK_ENTRY from the TASK_QUEUE and outputs both the entry and the new queue.
Counter_Interrupt:	Decrements the time delay of the first TASK_ENTRY in the TASK_QUEUE by the specified amount. If the first task must not then be delayed any longer, a flag is set to indicate this.

Waitlist Operation Descriptions (continued):

T3rupt: Extracts the first task from the TASK_QUEUE for execution. Sets a flag (ruptagn) to indicate if the next task is also to be executed immediately. In the AGC, T3RUPT is the program interrupt to service the Waitlist System when the TIME3 counter overflows.

Taskover: If the flag ruptagn is True, then T3rupt is reinvoked to process the task interrupt. Otherwise, no change is made to the TASK_QUEUE and the task-identifying output-address is EmtyAddr because no new task is to be executed.

Longcall: Because in the AGC the value of the Delta_T of a TASK_ENTRY has a maximum limit, one task at a time can be delayed longer than this limit using the Longcall operation. The constant time, CycleTime, has a value that is this maximum delay. The time input to Longcall is the total delay before the task is to be executed. Longcall uses Longcycle to recycle the delay until the delay interval is within the maximum limit. Taskcall is then invoked to enter the actual task invocation request.

Waitlist Operation Descriptions (continued):

Longcycle:	Decrements the time delay by the amount of CycleTime and enters *Longcall as a task to be executed* after the delay CycleTime. LongcallAddr is a constant ADDRESS with a value which identifies Longcall as a task.
Taskcall:	Makes the final Waitlist entry for the delayed task of Longcall, to be executed after the delay time when it is less then the CycleTime period.

OPERATION: b = Preceeds?(t_1,t_2);

WHERE b,b_1,b_2,b_3 ARE BOOLEANS;

WHERE t_1,t_2 ARE TIMES;

b = And(b_1,b_2) JOIN (b_1,b_2) = F_1(t_1,t_2);

b_1 = Notafter(t_1,t_2) COINCLUDE b_2 = F(t_1,t_2);

b_2 = Not(b_3) JOIN b_3 = Equal(t_1,t_2);

END Preceeds?;

DERIVED OPERATION: $time_2$ = Reverse($time_1$);

WHERE t IS A TIME;

Advance(Reverse(t),t) = Notime;

END Reverse;

OPERATION: t_3 = Regress(t_1,t_2);

WHERE t_1,t_2,t_3,t_4 ARE TIMES;

t_3 = Advance(t_1,t_4) COJOIN t_4 = Reverse(t_2);

END Regress;

HIGHER ORDER SOFTWARE, INC. • 843 MASSACHUSETTS AVENUE • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 661-8900

```
OPERATION:  e = New_Task_Entry(a,t);
     WHERE e IS A TASK_ENTRY;
     WHERE a IS AN ADDRESS;
     WHERE t IS A TIME;
WHEREBY e = Asgn_Task(EmtyTask,a,t);
END New_Task_Entry;
```

```
OPERATION:  q_o = Update_Waittime(q_i,t);
     WHERE t IS A TIME;
     WHERE q_i,q_o,q_1 ARE TASK_QUEUES;
     WHERE e_1,e_2 ARE TASK_ENTRYS;
     WHERE a IS AN ADDRESS;
q_o = Enter_Task(q_1,e_2) JOIN (q_1,e_2) = F_1(q_i,t);
q_1 = Discard_Task(q_i) COINCLUDE e_2 = S_2(q_i,t);
e_2 = New_Task_Entry(a,t) COJOIN a = F_3(q_i);
a = Task_Addr(e_1) JOIN e_1 = First_Task(q_i);
END Update_Waittime;
```

```
OPERATION:  (q_o,e) = Pop_First_Task(q_i);
     WHERE q_i,q_o ARE TASK_QUEUES;
     WHERE e IS A TASK_ENTRY;
q_o = Discard_Task(q_i) COINCLUDE e = First_Task(q_i);
END Pop_First_Task;
```

OPERATION: (q_o,overflow?) = Counter_Interupt(q_i,tinc);

WHERE q_i,q_o ARE TASK_QUEUES;

WHERE overflow? IS A BOOLEAN:

WHERE e IS A TASK_ENTRY;

WHERE tinc, t_1,t_2 ARE TIMES;

(q_o,overflow?) = F_1(q_i,t_2) COJOIN t_2 = F_2(q_i,tinc);

q_o = Update_Waittime(q_i,t_2) COINCLUDE WHEREBY overflow? = Notafter(t_2,Notime);

t_2 = Regress(t_1,tinc)

COJOIN t_1 = Delta_T(e)

COJOIN e = First_Task(q_i);

END Counter_Interupt;

```
OPERATION:  (q_o,ago,ruptagn) = T3rupt(q_i);

     WHERE q_o,q_i,q ARE TASK_QUEUES;

     WHERE ego,enew ARE TASK_ENTRYS;

     WHERE ago IS AN ADDRESS;

     WHERE ruptagn IS A BOOLEAN;

     WHERE past,delt,t ARE TIMES;

(q_o,ago,ruptagn) = F_1(q,ego,t) JOIN (q,ego,t) = F_2(q);
                     2  2           2  2

q_o = Update_Waittime(q,t)
                      2

     COINCLUDE ago = Task_Addr(ego)
                              2

     COINCLUDE WHEREBY ruptagn = Notafter(t,Notime);


(q,ego,t) = F_3(q,ego) JOIN (q,ego) = Pop_First_Task(q_i);
 2  2

WHEREBY (q,ego) = (q,ego) COINCLUDE t = F_4(q,ego);
         2  2

t = Regress(delt,past) JOIN (delt,past) = F_5(q,ego);


delt = F_6(q)  INCLUDE  past = Delta_T(ego);


delt = Delta_T(enew) JOIN enew = First_Task(q);


END T3rupt;
```

```
OPERATION:  (q_o,a,ruptagn_o) = Taskover(q_i,ruptagn_i);

     WHERE q_i,q_o ARE TASK_QUEUES;

     WHERE a IS AN ADDRESS;

     WHERE EmtyAddr IS A CONSTANT ADDRESS;

     WHERE ruptagn_i,ruptagn_o ARE BOOLEANS;

(1q_o,1a,1ruptagn_o) = T3rupt(1q_i)

     COEITHER WHEREBY (2q_o,2a,2ruptagn_o) = (2q_i,EmtyAddr,2ruptagn_i);


PARTITION OF (q_o,a,ruptagn_o,q_i,ruptagn_1) IS

     1(q_o,a,ruptagn_o,q_i,ruptagn_i)|ruptagn_i,

     2(q_o,a,ruptagn_o,q_i,ruptagn_i)|Not(ruptagn_i);

END Taskover;



OPERATION:  (q_o,a_o,t_o) = Longcall(q_i,a_i,t_i);

     WHERE q_i,q_o ARE TASK_QUEUES;

     WHERE a_i,a_o ARE ADDRESSES;

     WHERE t_i,t_o ARE TIMES;

     WHERE CycleTime IS A CONSTANT TIME;

(1q_o,1a_o,1t_o) = Longcycle(1q_i,1a_i,1t_1) COEITHER (2q_o,2a_o,2t_o) = Taskcall(2q_i,2a_i,2t_i);

PARTITION OF (q_o,a_o,t_o,q_i,a_i,t_i) IS

     1(q_o,a_o,t_o , q_i,a_i,t_i)|Preceeds?(Notime,Regress(t_i,CycleTime));

     2(q_o,a_o,t_o , q_i,a_i,t_i)|Notafter(Notime,Regress(t_i,CycleTime));

END Longcall;
```

```
OPERATION:  (q_o,a_o,t_o) = Longcycle(q_i,a_i,t_i);
     WHERE q_i,q_o ARE TASK_QUEUES;
     WHERE e IS A TASK_ENTRY;
     WHERE t_i,t_o ARE TIMES;
     WHERE CycleTime IS A CONSTANT TIME;
     WHERE a_i,a_o ARE ADDRESSES;
     WHERE LongcallAddr IS A CONSTANT ADDRESS;

q_o = F_1(q_i) COINCLUDE WHEREBY t_o = Regress(t_i,Cycletime)
     COINCLUDE WHEREBY a_o = a_i;

q_o = Enter_Task(q_i,e)
     COJOIN WHEREBY e = New_Task_Entry(LongcallAddr,CycleTime);

END Longcycle;


OPERATION:  (q_o,a_o,t_o) = Taskcall(q_i,a_i,t_i);
     WHERE q_i,q_o ARE TASK_QUEUES;
     WHERE e IS A TASK_ENTRY;
     WHERE a_i,a_o ARE ADDRESSES;
     WHERE t_i,t_o ARE TIMES;

q_o = F_1(q_i,a_i,t_i) COINCLUDE WHEREBY (a_o,t_o) = (a_i,t_i);

q_o = Enter_Task(q_i,e) COJOIN e = New_Task_Entry(a_i,t_i);

END Taskcall;
```

3.0 EXECUTIVE SYSTEM SPECIFICATION

The data types and service operations of the Executive System are overviewed below. The specifications themselves are contained in Sections 3.1 and 3.2 respectively.

Executive System Data Types:

- JOB_QUEUE: List of job invocation requests which are ordered according to their priorities.

- JOB_ENTRY: Information required to establish a job invocation request in the JOB_QUEUE. Includes proper identification of the job itself.

- PRIORITY: Indicates a relative position or importance in an ordering.

- TUPLE: A set of values which are ordered on the natural members.

- STACK: A "last in first out" list of values.

Executive System Service Operations:

- Findvac: Invoked to establish a job invocation request for a job that requires a VAC area (Vector Accumulator Temporay Storage Area) to be executed as soon as its priority is the highest in the JOB_QUEUE. (Corresponds to AGC Program entries: FINDVAC, and SPVAC, where SPVAC is a different implementation for shorter address references.)

Executive System Service Operations (continued):

- Novac: Invoked to establish a job request for a job without a VAC area.

- Change_Job: Invoked to restore the state of a currently executing job (at its same priority level) in the JOB_QUEUE, indicates which job in the queue is presently of highest priority. (Corresponds to AGC program entries: CHANG1 and CHANG2).

- Jobsleep: Invoked to "put a job to sleep" which is to remove from consideration for execution a job already established in the queue. Does not remove the job from the queue, "just puts it to sleep" for a while, until it gets "awoken" again. (Corresponds to AGC program entry: JOBSLEEP.)

- Jobwake; Returns a "sleeping" job back into consideration for execution. (Corresponds to the AGC program entry: JOBWAKE.)

- Priochng: Invoked to alter the priority of a job in the queue, with the consequence of a possible change in its relative position. (Corresponds to the AGC program entry: PRIOCHNG.)

- Endofjob: Invoked to remove a job from the executive system and release any storage allocated to it. (Corresponds to the AGC program entry: ENDOFJOB.)

3.1 Executive System Data Types

The data type specifications of the Executive System follow in this section. With each specification are included explinations of the function of the primitive operations and of the data-type axioms.

DATA TYPE: JOB_QUEUE(n);

PRIMITIVE OPERATIONS:

(job_queue_2,address) = Enter_Job(job_queue_1,job_entry,priority);

(job_entry,priority,address) = Top_Job(job_queue);

job_queue_2 = Discard_Job(job_queue_1,address);

(job_entry,priority) = Examine_Job(job_queue,address);

natural = Size (job_queue);

natural = Capacity(job_queue);

boolean = Equal(job_queue_1,job_queue_2);

AXIOMS:

WHERE n,n_1,n_2 ARE NATURALS;

WHERE $EmtyJobQ_n$ IS A CONSTANT JOB_QUEUE(n);

WHERE DummyJob IS A CONSTANT JOB_ENTRY;

WHERE DummyPrio IS A CONSTANT PRIORITY;

WHERE DummyAddr IS A CONSTANT ADDRESS;

WHERE $q_1,q_2,q_3,q,sq,nq,qi,sqi,nqi$ ARE JOB_QUEUES;

WHERE e_1,e,se,ne ARE JOB_ENTRYS;

WHERE p_1,p,sp,np ARE PRIORITYS;

WHERE adr,a_1,a,sa,na ARE ADDRESSES;

/*1*/ Top_Job($EmtyJobQ_n$) = (DummyJob,DummyPrio,DummyAddr);

/*2*/ Discard_Job($EmtyJobQ_n$,adr) = $K_{EmtyJobQ_n}(^1adr)$

OTHERWISE $K_{REJECT}(^2adr)$;

/*3*/ Examine_Job($EmtyJobQ_n$,adr) = ($K_{DummyJob}(^1adr)$,$K_{DummyPrio}(^1adr)$);

OTHERWISE $K_{REJECT}(^2adr)$;

DATA TYPE: JOB_QUEUE(n); (continued)

PARTITION OF adr IS

^{1}adr | adr = DummyAddr,

^{2}adr | adr ≠ DummyAddr;

/*4*/ Size($EmtyJobQ_n$) = 0;

/*5*/ Capacity($EmtyJobQ_n$) = n;

/*6*/ Equal($EmtyJobQ_{n_1}$, $EmtyJobQ_{n_2}$) = Equal(n_1, n_2);

/*7*/ Equal(q_2,q_3) = Equal(q_3,q_2);

/*8*/ Enter_Job(sqi,se,sp) = WHEREBY (q,a)

OTHERWISE K_{REJECT}(nqi,ne,np,nq,na);

PARTITION OF (sqi,se,sp,sq,sa) IS

(qi, e, p, q, a) | Size(sqi) < Capacity(sqi),

(nqi,ne,np,nq,na) | Size(sqi) ≥ Capacity(sqi);

/*9*/ Discard_Job(q,a) = qi;

/*10*/ Examine_Job(q,a) = (e,p);

/*11*/ q ≠ $EmtyJobQ_n$;

/*12*/ Size(q) = 1 + Size(qi);

/*13*/ Capacity(q) = Capacity(qi);

/*14*/ Equal(Enter_Job(q_1,e_1,p_1),(q,a)) =

Equal(q_1,qi) & Equal(e_1,e) & Equal(p_1,p);

DATA TYPE: JOB_QUEUE(n); (continued)

/*15*/ a ≠ DummyAddr;

/*16*/ IDENTIFY$_2^2$(Enter_Job(q,e_1,p_1)) ≠ a;

/*17*/ Top_Job(q) = (^{1}e,^{1}p,^{1}a) OTHERWISE Top_Job(^{2}qi);

PARTITION OF (qi,e,p,q,a) IS
^{1}(qi,e,p,q,a) | Higher?(p,IDENTIFY$_2^3$(Top_Job(qi))),
^{2}(qi,e,p,q,a) | -Higher?(p,IDENTIFY$_2^3$(Top_Job(qi)));

END JOB_QUEUE(n);

DATA TYPE: JOB_QUEUE(n); (continued)

Primitive Operation Descriptions:

(q_2,a) = Enter_Job(q_1,e,p);	Makes an entry into the JOB_QUEUE. The JOB_ENTRY is ordered in the queue according to the specified PRIORITY. An ADDRESS is returned as an identifier not of the job's location in the queue, but for random refference to the job while it is in the queue.
(e,p,a) = Top_Job(q);	Identifies which job in the queue is currently of highest priority.
q_2= Discard_Job(q_1,a);	Removes from the queue, the JOB_ENTRY identified by the given address. Will reject if no job is associated with the specified address.
(e,p) = Examine_Job(q,a);	Identifies from the queue, the job and its priority that is associated with the specified address.
n = Size(q);	A count of the net number of entries that have been made to the queue.
n = Capacity(q);	Indicates the maximum total number of entries that may be contained within the queue at one time.
b = Equal(q_1,q_2);	Determines if two queues are identical.

DATA TYPE: JOB_QUEUE(n); (continued)

Axiom Descriptions:

Axiom 1: Assures that DummyJob with its associated priority and identifier is always present in an EmtyJobQ.

Axiom 2: Discarding DummyJob using its identifier, DummyAddr, does not change the EmtyJobQ. However, any other address will reject because there can be no job associated with it.

Axiom 3: Only a DummyJob identified by DummyAddr exists in an EmtyJobQ to be examined.

Axiom 4: No jobs have been entered into a EmtyJobQ.

Axiom 5: The capacity of a JOB_QUEUE is determined by the subscript of the $EmtyJobQ_n$ that is was built from.

Axiom 6: Two $EmtyJobQ_n$'s are identical if their capacities are equal (as indicated by their subscribts).

Axiom 7: States that the test for equality is reflexive.

Axiom 8: States that Enter_Job will reject if the queue is already filled to capacity. A symbol is also defined in this statement for the case in which the queue is not previously filled to capacity.

Axiom 9: A job that is entered into the queue may be randomly removed by the Discard_Job operation.

Axiom 10: The address that is returned when a job is entered may be used to identify that job from within the queue.

DATA TYPE: JOB_QUEUE(n); (continued)

Axiom 11: An EmtyJobQ is never the same as a JOB_QUEUE in which an entry has been made.

Axiom 12: An entry made into a queue that is not filled to capacity will increase its size by one. Note that if DummyJob is entered via the Enter_Job operation, it will be treated as any other job and will increase the size of the queue.

Axiom 13: The capacity of a queue remains constant when jobs are entered (and when deleted as indicated by Axiom 9).

Axiom 14: The JOB_QUEUEs resulting from entries made into two non-filled queues will be identical if the original queues are identical and the two jobs are identical.

Axiom 15: The value of DummyAddr (which is associated with DummyJob) is never equal to the identifier of a job that has been entered into the queue. This is true even if the particular job that was entered was DummyJob. In this case, the axiom states that a different value than DummyAddr would be returned.

Axiom 16: Each different entry into the queue has a unique identifier.

Axiom 17: The process of entering a job into the queue will either place the new job at the front of the queue if it has the highest priority or place it in proper order so that repetitive applications of Discard_Job and Top_Job will find the new job in front of all jobs with lower priorities.

DERIVED OPERATION: job_queue_2 = Replace_Job(job_queue_1,job_entry,address);

WHERE q,sq,nq,qi,sqi,nqi ARE JOB_QUEUES;

WHERE e_1,e_2,e,se,ne ARE JOB_ENTRYS;

WHERE p,sp,np ARE PRIORITYS;

WHERE a_1,a_2,a,sa,na ARE ADDRESSES;

Replace_Job($EmtyJobQ_n$,e_1,a_1) = REJECT;

Enter_Job(sqi,se,sp) = WHEREBY (q,a) OTHERWISE K_{REJECT}(nqi,ne,np,nq,na);

PARTITION OF (sqi,se,sp,sq,sa) IS

(qi, e, p, q, a) | Size(sqi) < Capacity(sqi),

(nqi,ne,np,nq,na) | Size(sqi) $\geq$ Capacity(sqi);

Examine_Job(Replace_Job(q,e_1,a),a) = (e_1,p);

Equal(Replace_Job(q,e_1,a_1),Replace_Job(q,e_2,a_2)) = Equal(e_1,e_2) & Equal(a_1,a_2);

Size(Replace_Job(q,e_1,a)) = Size(q);

Capacity(Replace_Job(q,e_1,a)) = Capacity(q);

END Replace_Job;

The effect of the Replace_Job operation is that the resulting JOB_QUEUE has had replaced the JOB_ENTRY associated with the specified ADDRESS by the new JOB_ENTRY that is specified. No other changes occur. If no JOB_ENTRY has been associated with the specified ADDRESS, the operation will reject.

DATA TYPE: JOB_ENTRY;

PRIMITIVE OPERATIONS:

tuple = Job_Reg_Set(jobentry);

address = Loc&Bankset(jobentry);

address = Vac_Addr(jobentry);

boolean = Asleep?(jobentry);

$jobentry_2$ = Assgn_Job_Entry($jobentry_1$,tuple,$address_1$,$address_2$,boolean)

boolean = Equal($jobentry_1$,$jobentry_2$);

AXIOMS:

WHERE e,e_1,e_2 ARE JOBENTRYS;

WHERE EmtyJob IS A CONSTANT JOB_ENTRY;

WHERE reg IS A TUPLE;

WHERE loc,adr ARE ADDRESSES;

WHERE zzz IS A BOOLEAN;

/*1*/ Job_Reg_Set(Assgn_Job_Entry(e,reg,loc,adr,zzz)) = reg;

/*2*/ Loc&Bankset(Assgn_Job_Entry(e,reg,loc,adr,zzz)) = loc;

/*3*/ Vac_Addr(Assgn_Job_Entry(e,reg,loc,adr,zzz)) = adr;

/*4*/ Asleep?(Assgn_Job_Entry(e,reg,loc,adr,zzz)) = zzz;

/*5*/ Equal(e_1,e_2) = Equal(Job_Reg_Set(e_1),Job_Reg_Set(e_2))
& Equal(Loc&Bandset(e_1),Loc&Bandset(e_2))
& Equal(Vac_Addr(e_1),Vac_Addr(e_2))
& Equal(Asleep?(e_1),Asleep?(e_2));

DATA TYPE: JOB_ENTRY; (continued):

```
/*6*/     Job_Reg_Set( EmtyJob ) = REJECT;

/*7*/     Loc&Bandset( EmtyJob ) = REJECT;

/*8*/     Vac_Addr( EmtyJob ) = REJECT;

/*9*/     Asleep?( EmtyJob ) = REJECT;

END JOB_ENTRY;
```

Primitive Operation Descriptions:

t = Job_Reg_Set(j)	In the AGC, a job register set is a group of data cells set aside for each established job for use as temporary storage and status information. This is specified as a TUPLE.
a = Loc&Bankset(j)	In the AGC the cells LOC and BANKSET determine a unique address for a job starting location. This amounts to a unique identifier or name for a job, and is specified as an ADDRESS.
a = VAC_Addr(j)	An AGC VAC (Vector ACcumulator) area is a stack-like storage area. Vac_Addr indicates the starting address of the VAC area.

DATA TYPE: JOB_ENTRY; (continued)

b = Asleep?(j)	An AGC job that has been "put to sleep" will not be considered for execution regardless of its priority until it is "awakened." It does, however, retain its status as an established job. This is specified as a boolean-value attribute of a JOBENTRY.
j_2 = Assgn_Job_Entry (j_1,t,a_1,a_2,b)	(Assign attributes to a Job_Entry.) The JOBENTRY, j_1, is given the attributes t_1,a_1,a_2,b to produce j_2.

Axiom Descriptions:

Axioms 1-4:	The attributes assigned to an JOB_ENTRY are those that will be produced when it is examined.
Axiom 5:	Two JOB_ENTRYs are equal when all their attributes are equal.
Axioms 6-9:	An Empty_Job_Entry has no attributes.

A JOBENTRY is an information-storage device. It differs from a data structure in that as a data type it entails no internal structure. In implementation an internal structure is necessary. This, however, is not of concern at this layer of abstraction. A JOBENTRY corresponds to an established job in the AGC.

DATA TYPE: PRIORITY;

PRIMITIVE OPERATIONS:

boolean = Higher?(priority$_1$,priority$_2$);

boolean = Equal(priority$_1$,priority$_2$);

AXIOMS:

WHERE p,p_1,p_2 ARE PRIORITYS;

/*1*/ Equal(p,p) = True;

/*2*/ Equal(p_1,p_2) = Equal(p_2,p_1);

/*3*/ Entails(Equal(p_1,p_2)&Equal(p_2,p_3),Equal(p_1,p_3)) = True;

/*4*/ Higher?(p,p) = False;

/*5*/ Entails(Higher?(p_1,p_2)&Higher?(p_2,p_3),Higher(p_1p_3)) = True;

/*6*/ Entails(Higher?(p_1,p_2)&Higher?(p_2,p_1),Equal(p_1,p_2)) = True;

/*7*/ Higher?(p_1,p_2)!Higher?(p_2,p_1) = True;

END PRIORITY;

Primitive Operation Descriptions:

Higher?	A boolean function indicating whether or not the first input PRIORITY is of greatest importance (i.e., "higher priority") than the second.
Equal	Boolean function indicating the equality of two PRIORITYs.

Axiom Descriptions:

Axioms 1-3	Characterizes the equal OPERATION as an equivilance relation.
Axioms 4-7	Characterizes PRIORITYs as a totally ordered set.

```
DATA TYPE:  TUPLE;

PRIMITIVE OPERATIONS:

     tuple2 = Asgn_Item( tuple1,anytype,natural );

     anytype = Examine_Item( tuple,natural );

     boolean = Empty?( tuple );

     boolean = Equal( tuple1,tuple2 );

AXIOMS:

     WHERE  t,t1,t2  ARE TUPLES;

     WHERE  a,a1,a2  ARE OF SOME TYPE;  WHERE  ANYTYPE IS SOME TYPE;

     WHERE  n,n1,n2.n3,n4  ARE NATURALS;

     WHERE EmtyTuple  IS A CONSTANT TUPLE;

/*1*/     Empty?( EmtyTuple ) = True;

/*2*/     Empty?( Asgn_Item(t,a,n) ) = False;

/*3*/     Examine_Item( Asgn_Item(t,a,n3),n4 ) = 1a
          OTHERWISE  Examine_Item(2t,2n4);

PARTITION OF  (t,a,n3,n4)  IS
1(t,a,n3,n4) | n3 = n4,
2(t,a,n3,n4) | n3 ≠ n4;

/*4*/     Equal( Asgn_Item(t1,a1,n1),Asgn_Item(t2,a2,n2) ) =
               Equal(t1,t2) & Equal(a1,a2) & Equal(n1,n2);

/*5*/     Equal( EmtyTuple,EmtyTuple ) = True;

/*6*/     Equal( Asgn_Item(t,a,n),EmtyTuple ) = False;

/*7*/     Examine_Item( EmtyTuple,n ) = REJECT;

END TUPLE;
```

DATA TYPE: TUPLE; (continued)

Primitive Operation Descriptions:

t_2 = Assgn_Item(t_1,a,n)	Enter the value of a into t_1 at the ordering-position indicated by n to produce t_2.
a = Examine_Item(t,n)	Give a the value that was entered in t at ordering-position n.
b = Empty?(t)	Has t been assigned any values?
b = Equal(t_1,t_2)	Is t_1 identical to t_2?

Axiom Descriptions:

Axiom 1: The value of EmtyTuple is a TUPLE for which no entries have been made.

Axiom 2: Any TUPLE which has been assigned an entry is never equal to EmtyTuple.

Axiom 3: The value examined at an ordering-position will be the value that has been assigned there.

Axiom 4: The results of assignments to two TUPLEs are identical if the original TUPLEs are identical, the assigned values are equal, and the assigned positions are equal.

Axiom 5: Two instances of EmtyTuple are equal.

Axiom 6: EmtyTuple can never equal a TUPLE which has had assignments.

Axiom 7: No values have been entered into EmtyTuple and hence, none can be examined.

HIGHER ORDER SOFTWARE, INC. • 843 MASSACHUSETTS AVENUE • CAMBRIDGE, MASSACHUSETTS 02139 • (617) 661-8900

```
DATA TYPE:  STACK;

PRIMITIVE OPERATIONS:

     stack2 = Push( stack1,anytype );

     stack2 = Pop( stack1 );

     anytype = Top( stack );

     boolean = Equal( stack1,stack2 );

AXIOMS:

     WHERE  EmtyStack  IS A CONSTANT STACK;

     WHERE  ANYTYPE  IS SOME TYPE;

     WHERE  t  IS OF SOME TYPE;

     WHERE  s,s1,s2  ARE STACKS;

/*1*/     Top( EmtyStack ) = REJECT;

/*2*/     Top( Push(s,t) ) = t;

/*3*/     Pop( EmtyStack ) = REJECT;

/*4*/     Pop( Push(s,t) ) = s;

/*5*/     Equal(s1,s2) = Equal( Top(s1),Top(s2) ) & Equal( Pop(s1),Pop(s2) );

/*6*/     Equal( EmtyStack,Push(s,t) ) = False;

/*7*/     Equal( EmtyStack,EmtyStack ) = True;


END STACK;
```

DATA TYPE: STACK; (continued)

Primitive Operation Descriptions:

s_2 =	n(s_1,t);	Enters an item onto the stack.
s_2 =	(s_1);	Removes the top item from the stack.
t =	s);	Examines the top item on the stack.
b =	!(s_1,s_2);	Test to determine if two stacks are identical.

Axiom Descriptions:

Axi	No items exists in EmtyStack to examine.
Axi	The Push operation enters the specified item at the top of the stack.
Axi	No items exists in EmtyStack to remove.
Axi	Removing the last placed item returns the original value.
Axi -7:	Provide a deductive basis for evaluating the equality of two stacks. Two stacks are identical if all the items they contain are identical and in identical order. EmtyStack is never equal to a stack with an item entered, and two instances of EmtyStack are equal.

3.2 Executive System Operations

The specifications of all the Executive System operations are contained in this section. A description of the function of the operations is given below. The specifications follow.

Executive Operation Descriptions:

Findvac: Establishes a job in the JOB_QUEUE and allocates a VAC Area to the job. Will reject if no VAC Areas are available or if the JOB_QUEUE is filled to capacity. INVOKES: Novac2;

Novac: Establishes a job without allocating a VAC Area. Will reject if the JOB_QUEUE is full. INVOKES: Novac2;

Novac2: Enters a new job into the JOB_QUEUE. Will reject if the queue is full. INVOKED BY: Findvac,Novac; INVOKES: New_Job_Entry;

New_Job_Entry: Creates a new JOB_ENTRY from the specified attributes by assigning them to an EmtyJob. INVOKED BY: Novac2;

Change_Job: Replaces a specified JOB_ENTRY at the location given (in the relative position given by the priority it was entered with) and produces the location and JOB_ENTRY that has the highest priority.

Jobsleep: Puts to sleep the JOB_ENTRY at the specified location in the JOB_QUEUE and it assigns its Loc&Bankset according to the address specified. INVOKES: Go_To_Sleep,Asgn_Loc&Bankset;

Executive Operation Descriptions (continued):

Go_To_Sleep:	Puts a JOB_ENTRY into the state of "being asleep." INVOKES: Set_Sleep_State; INVOKED BY: Jobsleep;
Jobwake:	Awakens the JOB_ENTRY at the specified location in the JOB_QUEUE. INVOKES: Wake_Up;
Wake_Up:	Puts a JOB_ENTRY into the state of "being awake." INVOKES: Set_Sleep_State; INVOKED BY: Jobwake;
Set_Sleep_State:	Sets the sleep state of a JOB_ENTRY according to the boolean value specified. INVOKED BY: Go_To_Sleep,Wake_Up;
Asgn_Loc&Bankset:	Assigns the specified address to the Loc&Bankset of the JOB_ENTRY. INVOKED BY: Jobsleep;
Priochng:	Removes the JOB_ENTRY at the specified location in the JOB_QUEUE and reenters it with a new priority.
Endofjob:	Removes a JOB_ENTRY from the JOB_QUEUE and releases its VAC Area (if one was assigned) for further use.
New_Job_Yet?:	Determines if there is a JOB_ENTRY in the JOB_QUEUE with a higher priority than the one at the specified location.

```
OPERATION:  (q_o,s_o,locctr) = Findvac(q_i,s_i,newprio,newloc);

     WHERE q_i,q_o ARE JOB_QUEUES;

     WHERE s_i,s_o ARE STACKS;

     WHERE newloc,locctr,vacaddr ARE ADDRESSES;

     WHERE newprio IS A PRIORITY;

s_o = Pop(s_i) COINCLUDE (q_o,locctr) = F_1(q_i,s_i,newprio,newloc);

(q_o,locctr) = Novac2(q_i,newloc,vacaddr,newprio) COJOIN vacaddr = Top(s_i);

END Findvac;



OPERATION:  (q_o,locctr) = Novac(q_i,newprio,newloc);

     WHERE q_i,q_o ARE JOB_QUEUES;

     WHERE newloc,locctr,vacaddr ARE ADDRESSES;

     WHERE newprio IS A PRIORITY;

     WHERE NoVacAddr IS A CONSTANT ADDRESS;

WHEREBY (q_o,locctr) = Novac2(q_i,newloc,NovacAddr,newprio);

END Novac;
```

```
OPERATION:  (q_o,locctr) = Novac2(q_i,newloc,vacaddr,newprio);

     WHERE q_i,q_o ARE JOB_QUEUES;

     WHERE e IS A JOB_ENTRY;

     WHERE newloc,vacaddr ARE ADDRESSES;

     WHERE newprio IS A PRIORITY;

(q_o,locctr) = Enter_Job(q_i,e,newprio)

     COJOIN WHEREBY e = New_Job_Entry(EmtyTuple,newloc,vacaddr,False);

END Novac2;



OPERATION:  e = New_Job_Entry(v,a_1,a_2,b)

     WHERE e IS A JOB_ENTRY;

     WHERE v IS A TUPLE;

     WHERE a_1,a_2 ARE ADDRESSES

     WHERE b IS A BOOLEAN;

WHEREBY e = Asgn_Job_Entry(EmtyJob,v,a_1,a_2,b);

END New_Job_Entry;
```

```
OPERATION:  (q_o,e_o,loc_o,fixloc) = Change_Job(q_i,e_i,loc_i);

     WHERE q_i,q_o ARE JOB_QUEUES;

     WHERE e_i,e_o,e_1 ARE JOB_ENTRIES;

     WHERE loc_i,loc_o,loc_1,fixloc ARE ADDRESSES;

     WHERE ovfind IS A BOOLEAN;

     WHERE qtop IS (JOB_ENTRY,PRIORITY,ADDRESS);

(q_o,e_o,loc_o,fixloc) = F_2(q_1) JOIN q_1 = Replace_Job(q_i,e_i,loc_i);

WHEREBY q_o = q_1  COINCLUDE (e_o,loc_o,fixloc) = F_3(q_1);

(e_o,loc_o,fixloc) = F_4(e_1,loc_1) JOIN (e_1,loc_1) = IDENTIFY^3_1,3 (qtop)

     JOIN qtop = Top_Job(q_1);

WHEREBY (e_o,loc_o) = (e_1,loc_1) COINCLUDE fixloc = Vac_Addr(e_1);

END Change_Job;



OPERATION:  q_o = Jobsleep(q_i,locctr,newloc);

     WHERE q_i,q_o ARE JOB_QUEUES;

     WHERE e_1,e_2,ezz ARE JOB_ENTRYS;

     WHERE locctr,newloc ARE ADDRESSES;

     WHERE ep IS (JOB_ENTRY,PRIORITY);

q_o = Replace_Job(q_i,ezz,loc) COJOIN ezz = Go_To_Sleep(e_2)

     COJOIN e_2 = Asgn_Loc&Bankset(e_1,newloc)

     COJOIN e_1 = IDENTIFY^2_1(ep)

     COJOIN ep = Examine_Job(q_i,locctr);

END Jobsleep;
```

```
OPERATION:  e_o = Go_To_Sleep(e_i);

     WHERE e_i,e_o ARE JOB_ENTRYS;

WHEREBY e_o = Set_Sleep_State(e_i,True);

END Go_To_Sleep;



OPERATION:  q_o = Jobwake(q_i zzloc);

     WHERE q_i,q_o ARE JOB_QUEUES;
     WHERE ezz,eup ARE JOB_ENTRYS;
     WHERE zzloc IS AN ADDRESS;
     WHERE ep IS (JOB_ENTRY,PRIORITY);

q_o = Replace_Job(q_i,eup,zzloc)

     COJOIN eup = Wake_Up(ezz)
     COJOIN ezz = IDENTIFY_1^2(ep)
     COJOIN ep = Examine_Job(q_i,zzloc);

END JobWake;



OPERATION:  e_o = Wake_Up(e_i);

     WHERE e_i,e_o ARE JOB_ENTRYS;

WHEREBY e_o = Set_Sleep_State(e_i,False);

END Wake_Up;
```

```
OPERATION:  e_o = Set_Sleep_State(e_i,asleep?)

     WHERE e_i,e_o ARE JOB_ENTRYS;

     WHERE v IS A TUPLE;

     WHERE loc,va ARE ADDREESES;

     WHERE asleep? IS A BOOLEAN;

e_o = Asgn_Job_Entry(e_i,v,loc,va,asleep?)

     COJOIN v = Job_Reg_Set(e_i)

     COJOIN loc = Loc&Bankset(e_i)

     COJOIN va = Vac_Addr(e_i);

END Set_Sleep_State;



OPERATION:  e_o = Asgn_Loc&Bankset(e_i,loc);

     WHERE e_i,e_o ARE JOB_ENTRYS;

     WHERE reg IS A TUPLE;

     WHERE loc,va ARE ADDRESSES;

     WHERE zz IS A BOOLEAN;

e_o = Asgn_Job_Entry(e_i,reg,loc,va,zz)

     COJOIN reg = Job_Reg_Set(e_i)

     COJOIN va = Vac_Addr(e_i)

     COJOIN zz = Asleep?(e_i);

END Asgn_Loc&Bankset;
```

```
OPERATION:  (q_o,loc) = Priochng(q_i,locctr,newprio);
     WHERE q_i,q_o,q_1 ARE JOB_QUEUES;
     WHERE e IS A JOB_ENTRY;   WHERE locctr IS AN ADDRESS;
     WHERE newprio IS A PRIORITY;
     WHERE ep IS (JOB_ENTRY,PRIORITY);

(q_o,loc) = Enter_Job(q_1,e,newprio) COJOIN (q_1,e) = F_1(q_i,locctr);

q_1 = Discard_Job(q_i,locctr) COINCLUDE e = F_2(q_i,locctr);

e = IDENTIFY_1^2(ep) JOIN ep = Examine_Job(q_i,locctr);

END Priochng;
```

```
OPERATION: (q_o,vs_o) = Endofjob(q_i,vs_i,loc);
     WHERE q_i,q_o ARE JOB_QUEUES;
     WHERE vs_i, vs_o ARE STACKS;
     WHERE e IS A JOB_ENTRY;
     WHERE loc,va ARE ADDRESSES;
     WHERE ep IS (JOB_ENTRY,PRIORITY);
     WHERE NovacAddr IS A CONSTANT ADDRESS;

q_o = Discard_Job(q_i,loc) COINCLUDE vs_o = F_1(q_i,vs_i,loc);

vs_o = F_2(vs_i,e) COJOIN e = F_3(q_i,loc);

e = IDENTIFY_1^2(ep) JOIN ep = Examine_Job(q_i,loc);

WHEREBY 1vs_o = 1vs_i COEITHER 2vs_o = F_4(2vs_i,2e);

PARTITION OF (vs_o,vs_i,e) IS
  1(vs_o,vs_i,e)|Vac_Addr(e) = NovacAddr,
  2(vs_o,vs_i,e)|Vac_Addr(e) ≠ NovacAddr;

2vs_o = Push(2vs_i,va) COJOIN va = Vac_Addr(2e);

END Endofjob;
```

```
OPERATION:  njy = New_Job_Yet?(q,loc);

     WHERE q IS A JOB_QUEUE;

     WHERE toploc,loc ARE ADDRESSES;

     WHERE qtop IS (JOB_ENTRY,PRIORITY,ADDRESS);

     WHERE njy IS A BOOLEAN;

njy = Equal(loc,toploc) COJOIN toploc = F1(q);

toploc = IDENTIFY3 3(qtop) JOIN qtop = Tob_Job(q);

END New_Job_Yet?;
```

Section V

A HIGHER ORDER MACHINE (HOM) FOR HIGHER ORDER SOFTWARE (HOS)

by

W. Heath

TABLE OF CONTENTS

1.0 INTRODUCTION

The availability of low-cost digital hardware has now motivated the use of automatic processing in applications that have much greater complexity than those previously feasible. Because of the degree of processing power that is required for these new applications, it is becoming impossible for existing single-processor architectures to meet the time constraints that are imposed. Single-processor systems simply do not operate fast enough to perform all of the required time-critical tasks. Other than increasing actual logic-operation speeds, these time constraints can be met by organizing the hardware so that more than one operation can be performed at a time. Many such "multiprocessing" architectures have been proposed, but by far the greatest difficulty encountered with each design is the software control of the many independent operations which occur simultaneously.

The theory of Higher Order Software (HOS) [HAM76a,b,c] provides a solution to the problem of specifying concurrent, asynchronous processes. HOS defines the interactions of data, rather than operations on hardware, as do most common software-definition techniques. As a result, an HOS specification provides the information necessary to execute software with a maximum degree of simultaneous operation. Using HOS it is possible to maintain complete asynchronous control of the processing hardware while minimizing storage use through dynamic memory allocation.

This report presents the Higher Order Machine concept as an implementation of systems that are specified according to the HOS methodology. The Higher Order Machine (HOM) is a fully distributed, modular, asynchronous multiprocessing system. It is totally reconfigurable and has the potential capability of performing any number of operations simultaneously.

HIGHER ORDER SOFTWARE, INC. · 843 MASSACHUSETTS AVENUE · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 661-8900

2.0 MULTIPROCESSING SYSTEM REQUIREMENTS

In Chapter 1, multiprocessing was discussed as a solution to the demand for greater system throughput. Much more is required of a multiprocessing system to fulfill the needs of the continually expanding applications for automatic processing. This chapter addresses some of the more important constraints that confront operational multiprocessing systems. Included in the discussion are the need for greater reliability in "on-line" applications, the ever-present economic demand for cost-effectiveness, the de-emphasis of hardware structure in a "function-first" approach to system design, a recognition of the need to "predict the unpredictable" and provide for a reconfiguration capability during system development, and an ending summary of the impact of these requirements on multiprocessing system structure.

2.1 Reliability

As automatic processing costs have decreased, more and more of the operation-critical functions of applications systems have been made dependent upon the reliable performance of computational equipment. Electronic component failure rates have been progressively decreasing as the technology has improved, but these reliabilities are not sufficient for the safety- and life-critical functions of many real-time applications. In these critical applications, it is necessary to provide fault-tolerant capabilities in the processing system to achieve acceptable reliability. This is readily incorporated into multiprocessing systems because alternate system configurations can be made available in the event of a hardware component failure. Thus, "fail-soft" capabilities, where system performance may be degraded by failures but not interrupted completely, can be built into the system without the high cost of "back-up" redundancy. If the execution-time advantages of multiprocessing are to benefit these highly safety-sensitive applications, then the necessary reliability must be provided within the system at reasonable costs.

2.2 Cost Effectiveness

A multiprocessing system would not be economically viable unless it provided a cost reduction over an equally powerful aggregate of simplex computers. This cost saving must come from an increase in hardware utilization during system operation. For example, increasing parallel-processing power without intensifying memory usage will provide no resultant cost improvement because processing costs are proportionally increased. A savings would then be derived only when more use of the same storage space can be made by multiprocessing, therefore reducing the cost of memory use. It is also possible to improve the usage of active processing hardware as well. In a simplex system, a large percentage of the active hardware is always idle because the system is generally capable of performing only one of the operations available to it at any instant. However, a multiprocessing system can be given the capability of using single-function hardware units independently of each other. Distributing the processing logic in this manner will improve the active hardware utilization in the system.

The cost-effectiveness of a multiprocessing system can be further improved by relaxing the speed requirements of individual logic operations. Overall throughput improvements can still be obtained even if each operation is not as fast as would be necessary in a simplex system. Thus expensive high-speed logic is not required for good system performance. What the multiprocessing concept promises then in economic terms is a means to achieve the performance required with less hardware and cheaper components.

2.3 "Function-First" Design Approach

Historically, the design and maintenance of computer hardware represented the greatest cost of an automatic processing system. As a result, hardware operation dictated the structure of the system. Software developed as the method by which a fixed machine was made to perform in a desired manner. However, with current

hardware cost reductions and the increased complexity of present-day applications, software specification has begun to consume a major portion of the system cost for both design and maintenance. It used to be true that orientation of the system development to the physical machine, rather than to the function it was performing, caused an overriding importance to be placed on the speed efficiency of the code produced for it. But it is becoming more and more apparent that the overall efficiency and reliability of the system is more dependent on a unified system structure than on the cleverness of scattered sequences of programming code.

It is now necessary that a system be analyzed according to the function it performs. Once the function of the system has been determined and completely specified, it is then possible to choose the best configuration of hardware and software to implement that function. This can range anywhere between the extremes of a complete software solution on existing "fixed" hardware and a total hardware solution in which the whole function is hardwired. From this general perspective, there is no categorical difference between a simplex system and a multiprocessing system. Instead, a uniprocessor is a special case of the many possible types and configurations of processing units, communication busses, and storage units. Thus, with this complete set of hardware configurations to draw from, the best hardware/software solution can be chosen to perform the desired system function as specified.

2.4 Developmental Reconfigurability

It is generally impossible to complete a system design and have it perform as required the first time. Even the requirements often cannot be completely specified beforehand. Usually too much has to be learned about the function of a system during its design for this to be accomplished. As a result, changes to a system can always be expected throughout its development and indeed throughout its life cycle as its requirements evolve and its use matures. For these reasons, an effective processing system must be easily reconfigured to adapt to inevitable modifications.

Reconfigurability is also necessary in a multiprocessing system to facilitate molding the physical hardware to the functional requirements of the applications system. This was discussed in the previous section as the "function-first" approach to systems design. The multiprocessing system architecture would not then be one machine, but a class of machines from which the best was chosen for the particular application. Economic considerations dictate that the chosen hardware configuration be assembled without costly individual redesign. A multiprocessing architecture composed of discrete modules with well-defined interfaces will fulfill these requirements. For reconfiguration, modules could then be added or removed according to the resource-allocation analysis of the function the system must perform. With this type of flexible architecture, only the least amount of hardware necessary need be incorporated into the system. Thus the size, weight, and cost of the hardware can be minimized with the least design effort.

2.5 Summary of Processing System Requirements

This chapter has outlined some new requirements for computing machinery in the face of a redirection in the emphasis of system development from operational hardware to system function. These requirements may be summarized as follows:

a. Low-cost reliability and fault tolerance should be available to a multiprocessing architecture design through dynamic reconfiguration capabilities.

b. The greatest intensity of use produces the greatest return for investment in hardware. This can best be accomplished by a distributed system in which the large majority of the hardware components can be simultaneously active.

c. The application function, not the hardware operations, should dictate the structure of the system.

d. The hardware should be tailored to the functional requirements of the system through a modular reconfiguration capability.

These considerations must be incorporated into the design of processing systems if the trend for applications to become more and more real-time oriented and safety critical is to continue, and if these systems are to become economically realizable. The chapters following examine how fully these requirements are fulfilled by existing multiprocessing system designs and how these requirements are implemented within the Higher Order Machine system.

3.0 EXTANT MULTIPROCESSING MACHINE DESIGNS

This chapter examines current multiprocessing concepts in the face of the requirements outlined in Chapter 2. The basic classes of multiprocessing machines that are discussed are the special-applications machines and the general-purpose multiprocessing systems.

3.1 Special-Applications Machines

The three special-purpose parallel machines that are described--the array processor, the associative processor, and the pipeline processor--are not intended as all-purpose computational systems. Instead, each is designed to perform efficiently a particular function which could be executed by a general-purpose machine but would take less time using the special hardware.

3.1.1 The Array Machine

The array machine consists of a bank of processors, all of which respond simultaneously to one sequence of instructions. The most elaborate operational machine is the ILLIAC IV which has a set of 64 processing elements [THU75]. These machines are very effective in dealing with such data types as n-tuples, vectors, and arrays, which have intrinsic parallelism in their operation sets.

3.1.2 Associative Processors

Like the array machine, the associative processor has a bank of processing elements which are controlled by a single instruction sequence, but greater power is built into each of its processing elements. The associative-processor elements are content addressable. This means that they have the capability of responding individually to an instruction depending on the content of their registers. This characteristic makes the machine very useful for data-base operations, such as searching for particular contexts or sorting data by its content. Banks of these content-

addressable elements, which are constructed with comparison logic only, are called content-addressable memory or associative memory. In using content-addressable memory, the actual physical location of data can be neglected and only the structure of the data need be of concern.

3.1.3 Pipelined Machines

Pipelining is a processing concept in which computation is partitioned into sequential stages. Data from one stage is passed on to the next for further processing. With such an organization, concurrent operation can occur because execution may proceed simultaneously with different data at each stage. As a result of its structure, effective use of pipelining is limited to specialized processing of serial data streams.

3.1.4 Summary of Special-Purpose Machines

Although the special-purpose hardware organizations have restricted applications, they do well what they are intended to do. Indeed, random-access memory is special-purpose hardware which can only store and retrieve data. Similarly, these hardware modules should be thought of as possible additions to a multiprocessing system that has a larger intended scope.

3.2 General-Purpose Multiprocessing Systems

There are two types of concurrent computation in general-purpose multiprocessing: (1) simultaneous execution of multiple tasks and (2) parallel execution of operations within a single task. This division will aid the evaluation of the capabilities of the systems designed to perform multiprocessing. In what follows, four hardware configurations are discussed:

1. the Multiprocessor,
2. the Computation Net Machine,
3. the Data-Flow Machine, and
4. the Single-Assignment Machine.

3.2.1 The Multiprocessor

Basic Multiprocessor

The multiprocessor is the most commonly discussed multiprocessing configuration and, in fact, is the only one of those examined in this section that is operational. This configuration is a multi-CPU assembly that is bootstrapped from the common simplex system. The basic multiprocessor system has a set of independent processing units which share a common memory space. Conceptually, each processor within the machine can perform independent operations for the same or for different tasks. However, because the processors share the same memory address space, they must contend for access to the same physical memory hardware. This creates a problem that can severely cripple the performance of the whole system. Because a storage module can service only one memory access at a time, it is desirable to interleave the memory address space of the system among several storage modules. But if more than one processor tries to access the same storage module, all but one must wait to be serviced. Depending on the numerical ratio of storage modules to processors, this will cause some percentage of the execution time to be lost to memory contention. Within a certain effective limit, this loss can be reduced by increasing the number of storage modules with respect to the number of processors (for a constant storage space). However, this is not a free improvement and will proportionally increase the total system costs attributed to passive storage resources and thus reduce the cost-effectiveness of the whole system.

Enhanced Multiprocessor

Elaborations of the basic multiprocessor configuration have been developed to reduce the processing time lost to memory contentions. Usually this has consisted of some combination of storage duplication and individual cache memory space. Storage duplication is simply the process of keeping identical copies of the most used portions of memory in different storage modules to increase their accessability, whereas a cache is a small amount of private memory assigned to each processor. By using individual cache

memory, it is possible to reduce the total number of accesses to main memory. This is done by maintaining a working copy of a section of main memory in each cache or by storing duplications of low-level routines or macro-expansions within each processor [SMI73]. While both cache memory and storage-duplication techniques will serve to decrease memory contention, their use will also degrade the cost-effectiveness of the system. There are two contributing factors. First, the isolation of some of the memory space in caches and the replication of portions of the memory contents will reduce the intensity of memory usage. Secondly, the software overhead incurred by the initiation of a parallel path will be greatly expanded, and as a result, the degree of fine-grain concurrency must be limited to levels for which parallelism is profitable despite the overhead. Both of these side effects are contrary to the concepts which motivated the development of multiprocessing systems, i.e., to provide a cost savings over an equally powerful network of autonomous computers.

Multiprocessor Analysis

The multiprocessor is a logical adaptation of conventional computer architectures to multiprocessing capabilities. It has the advantages of modularity, in terms of processing units, memory units, and busses, and of possible fault-tolerant dynamic reconfiguration capability with internal triad redundancy. However, it has severe limitations in meeting the requirements outlined in Chapter 2. Parallelism in the multiprocessor is mostly limited to simultaneous execution of different tasks because of the expense needed to initiate new parallel paths, the memory access conflicts, and the difficulty of manually programming the dynamic interaction of the processing units. A certain amount of pipelining is possible within an individual processor, but this is limited by storage-address dependencies and instruction dependencies. The multiprocessor primarily exhibits centralized control and suffers generally from low-usage intensity because each processor must execute instructions sequentially. This coarsely organized modularity also affects the costs of dynamic reconfiguration because, in the event

that a processing unit exhibits a hardware fault, the use of the complete processor must be discontinued.

The most critical deficiency of the multiprocessor from a "function-first" systems viewpoint is its fixed architectural structure. The possibility of optimizing the hardware to the application function without costly redesign is thus eliminated. This can be alleviated somewhat by "microprogrammable" and "nanoprogrammable" architectures which allow redefinition of different levels of hardware control, but such a system still derives its structures from a predetermined, fixed hardware. An organization of this nature is contrary to the concept of allowing the function to determine the extent and form of the hardware. The architecture of the multiprocessor reflects its bootstrapped development from a fixed single-processor computer and exhibits the similar necessity of limiting software definition to the "machine domain" instead of the "function domain."

3.2.2 The Computation-Net Machine

Net-Machine Structure

The core of the computation-net machine [SYL75] is a hardware array of arithmetic microprocessors (AMPs), each of which can take up two input operands and an operator and output one operand as the result of the indicated operation. The net machine has different storage units for instructions and operand values--the instruction memory (IM) and the operand memory (OM) respectively. The AMPs can traffic among themselves and with the IM and OM via a set of operational registers. A sequence and control unit (SCU) directs operations and traffic within the system. The SCU interprets instructions from the IM, fetches and stores operands in the OM, and organizes the operations performed by the AMPs, all via a queue and bus system.

Arithmetic Computation Nets

Operation of the net machine is based on the concept of arithmetic computation nets. A net is the decomposition of an arithmetic

expression into an input/output mapping of primitive operations which can be performed by an AMP. A net composition of the expression, e = (a+b)*(c-d), for example, is illustrated in Figure 3.2.2-1.

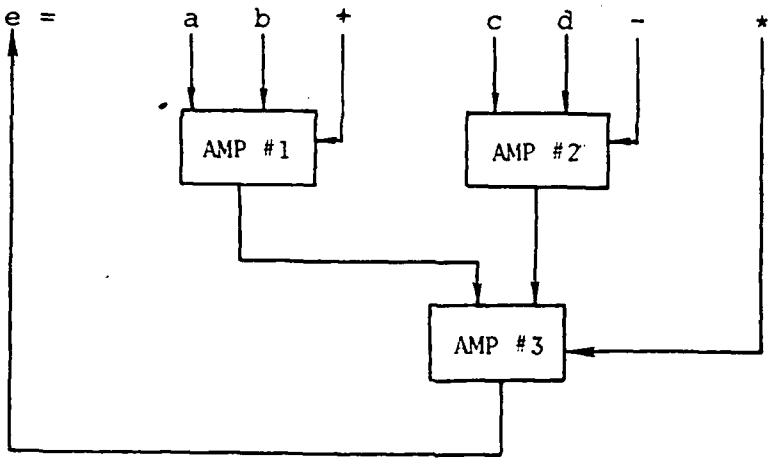


Figure 3.2.2-1
Arithmetic Computation Net for e = (a+b)*(c-d)

In a compilation process from a language used to define sequential arithmetic computation, a control code and instructions are generated for each net and stored in the IM. The control code is used to organize and link the AMP operands and to synchronize operand traffic.

Net-Machine Operation

Each net executes within the array of AMPs. Net execution is completely synchronous and is specified during compilation. Concurrent computation can take place for independent operations such as AMP #1 and AMP #2 in the example. AMP #3 could not execute until both AMP #1 and AMP #2 had produced their output, but if the net were larger, there could possibly be other operations executing concurrently with AMP #3. The size of the nets, and

therefore the concurrent capability of the computation-net machine is limited by the number of AMPs in the array because an AMP can be assigned only one operation in each single net execution. The net size is also limited by program-control logic since the language is strictly sequential and all transfers must separate nets. Address dependencies must also separate nets because all operand fetches must be made before net execution is initiated. When an address is computed dynamically, its operand fetch must be made in the beginning of the next net cycle. Through simulation [SYL75], it was found that ten AMPs in the array were typically the most that the logical programming restrictions or net size would allow.

Net-Machine Analysis

The arithmetic computation-net multiprocessing system is strictly a single-task machine that can have a concurrency of up to approximately ten simultaneous operations. However, there is no reason that more than one net processor could not share the same memory space and therefore have the capability of executing multiple tasks simultaneously. Although such a configuration would have the same structural deficiencies as the multiprocessor, the internal parallelism of each processor would be increased significantly. The extent of overhead for initiation of a parallel process might also be reduced because each net could be treated as a logical machine instruction.

3.2.3 The Data-Flow Machine

Data-Flow Language Concept

The Data-Flow Machine was designed to execute Data-Flow programs [RUM75]. Data-Flow programs are composed of procedures in the conventional sense, but Data-Flow procedures are defined in a particular manner and are designed to specify only deterministic computations. The Data-Flow Language is based on the concept that a datum is an autonomous entity, called a token, which is created by some particular operation and used or consumed by

another. A Data-Flow procedure is specified as a directed graph in which the nodes are operations and the directed links are one-way channels over which tokens travel. An operation executes (or fires) when all its argument tokens have appeared at its input links. It then "consumes" its input tokens and subsequently emits its result tokens at its output links. All operations in a Data-Flow procedure are asynchronous and depend only on the presence of input. A procedure is initiated by the presence of procedure argument tokens at its input and is completed when all the result tokens are produced at its output. One procedure can invoke another by use of an application mechanism for which the invoked procedure is considered an operation node at the outer level. The set of Data-Flow operations include conventional function operators as well as special mechanisms for flow control such as switches, unions and duplication-branch nodes. A Data-Flow procedure is basically a flow diagram something akin to a model-train layout. In this analogy, data tokens would be engines which carry values, operation nodes are junctions which can alter the token in some specified manner at a particular point in the line, and a control node is either a switch which operates from a signal token from another line, a duplication branch which sends out another duplicate train along a second line, or a union junction which meshes two sets of lines into one set.

Data-Flow Machine

The hardware architecture of the Data-Flow Machine is organized around a set of activation processors which implement the computation of a Data-Flow procedure. An activation processor operates on a procedure after it has been invoked until no more computation can be performed because of outstanding procedure calls. At this time, the procedure is deactivated and the activation processor takes up the computation of a newly activated procedure. When results become available from invoked procedures, the dormant procedure that had made the invocations is reactivated and computation is continued. Actual computa-

tion is performed in an activation processor by an execution pipeline. The execution pipeline consists of a parallel set of functional units in series with modules which control the pipeline operation. The functional units perform the primitive Data-Flow operations. Several operations may be executed concurrently by different stages of the pipeline, and, if loaded, the functional units can operate simultaneously. Concurrency of operation in the Data-Flow Machine is realized through simultaneous execution of multiple activation processors and through the overlap of functional processing within the execution pipeline of each individual activation processor.

Data-Flow Analysis

The Data-Flow Language represents a significant step above conventional programming toward the specification of software as a function instead of as directed operations on a machine. Data-independent operations, where the input to one does not depend on the output of the other, are intrinsically identified in the language. Thus parallelism is automatically detectable at any level. Side-effects cannot exist also because all data paths must be identified. However, the Data-Flow Language, as it is, is not suited to system specification because it is very low-level and difficult to use. This defficiency may possibly be overcome by the development of a suitable high-level language (perhaps similar to the Single-Assignment Programming Language discussed in the next section). Of more serious consequence is the fact that data links are not uniquely identified outside of the procedure in which they exist. This results in a less distributed machine than might possibly be developed. In the Data-Flow Machine, all operations must be executed within the "umbrella" of their procedure activation, causing artifical limitations on the size of procedures and unnecessary overhead in activating and deactivating procedures. The Data-Flow Language is also limited in the definition of its data types. The Data-Flow Procedures as described are more implementations of a calculation in terms of given

data types than the specification of a function. Implementing a particular calculation instead of specifying a function to be performed eliminates the selection of other possibly more efficient implementations. The Data-Flow Language does not specify a functional hierarchy but instead defines the lowest-level calculations to be performed. Thus there exists no facility for recovery from invalid inputs to a module and no inherent priority structure to guarantee that all the internal procedures of one module can interrupt all the internal procedures of another module of lower priority. Finally, no provision is made in the Data-Flow Language for the existence or passage of time or for the specification of non-determinate functions such as I/O operations and external events. These are all necessary functions that must be considered in real-life systems.

3.2.4 Single-Assignment Programming Concept

The concept of single-assignment programming was developed as a possible solution to the problem of organizing a multiple system of independent processors to perform in concert without interference of operation. To achieve this, it was recognized that each new value that is generated by a processor must have a unique identification. Thus, a variable must be assigned a value only a single time. Adherence to this constraint provides automatic detection of all possible parallelism in the execution of a specified computation. This is apparent because any two specified operations that are simultaneously ready for execution may be executed in parallel. An instruction may be performed just as soon as all the variables that it uses have been defined. Hence, there is no programmed "flow of control;" the sequence of instruction execution is determined by flow of the generation and subsequent use of data. The single-assignment programming language, SAMPLE (for Single-Assignment Mathematical Programming LanguagE), and a simulated machine-concept to execute it were developed by Chamberlin [CHA71].

Single-Assignment Programming Language

There are two data types in SAMPLE, real numbers and tuples, consisting of ordered sets of numbers or other tuples. Standard expressions and assignments permit the specification of arithmetic computation. There exists a conditional assignment statement with a boolean-valued predicate to allow programmed control of the computational flow, and there are two types of iteration control for either simultaneous or necessarily-sequential computation. Simultaneous iteration is used in replicated instructions on tuples, and sequential iteration is supported by a looping mechanism. Programs in SAMPLE may have a block structure through which variable-naming scope is controlled, but memory allocation will not be affected because computation flow is independent of instruction listing.

Single-Assignment Machine Structure

The hardware system has three passive storage units -- an instruction store, a data store, and a ready list. The instruction store contains the program in primitive hardware operations. Each operation calculates one output operand; the status of each input operand is maintained in the instruction of a "ready" flag which indicates the existence of its value. The data store contains the user data. A data cell has provision for a value as well as a pointer to the instruction which uses that value. If more than one instruction uses the value, then they follow the first in a linked list. If the variable is a tuple, then its value is itself a pointer to a contiguous set of cells which hold the tuple values. The ready list contains copies of all instructions that are ready to be executed because all their input operands have been defined. Active logic is contained in a set of independent processing units which execute in parallel operation.

System Operation

Computation is performed in instruction-execution cycles in which each processor repeats the following sequence of actions [SYL75]:

1. The processor fetches from the ready list an instruction which is ready to be executed.

2. The processor fetches from the data store the input operands of the instruction and performs the indicated operation on them.

3. The processor writes the resulting output operand into the data store. In the same storage-access cycle, it obtains the pointer to an instruction which is waiting for the newly-ready cell, if any.

4. The processor follows the linked list of instructions which are waiting for the newly-ready data cell. For each such instruction, it does the following:

    a. It turns on the ready bit of the newly-ready operand.

    b. If all ready bits are now on, it copies the instruction into the ready list.

    c. It obtains the link to the next instruction on the waiting list.

Iteration and function invocations are handled by dynamically generating new instructions during execution. In a tuple instruction copies of each instruction are replicated for each element of the tuple. Function invocations are made by filling in the input variable "names" in the proper locations in a template of the function instructions. A new copy of all the instructions of a loop body are generated with each execution of the loop to preserve single assignment. A concept of "levels of readiness" must be introduced in the execution of a loop, because all instructions in the currently executing copy of the loop must have been performed before their output variables may be referenced by external instructions.

Single-Assignment Analysis

Single-assignment programming and Data-Flow layouts are very similar in that they both express low-level operations on data. This is a significant departure from sequential programming which expresses non-unique operations on hardware components. However,

a hierarchical functional structure is still lacking. In the single-assignment techniques, this results in the inability to determine when the storage for data may be reallocated. Without a functional hierarchy, each data reference cannot be identified, and it is therefore impossible to determine when all references have been made. As with the data-flow techniques, timing and response to non-deterministic events are not provided for in the single-assignment language.

3.3 Summary of Extant Multiprocessing Concepts

In examining the current multiprocessing system concepts, the need for an adequate means of expressing functional relationships becomes apparent. The Single-Assignment and Data-Flow techniques are less machine oriented than the sequential programming of the multiprocessor or the net machine. None of these software techniques, however, is able to support the execution of a completely distributed, asychnronous system. The next chapter will establish the methodology of Higher Order Software as the solution to this problem.

4.0 SPECIFICATION OF CONCURRENT PROCESSES

Before the solution to a problem has been defined, it is impossible to determine how this solution can best be implemented for actual execution. This is the reason why the problem must be specified in a manner that is totally independent of any particular implementation. Higher Order Software (HOS) makes possible just such a problem description. HOS is a formal methodology for the specification of reliable systems that include components of hardware, firmware, software, and humanware and of the dynamic environment within which these systems reside. It is a formal theory based on a set of axioms that define a functional hierarchy for complete and consistent computable systems. The axioms formalize the interfaces, functional influences, and internal control of the system. The HOS theory is detailed in [HAM76a,b,c]; the language for HOS system specification, AXES, is described in [HAM76c].

Higher Order Software provides a solution to the problem of specifying concurrent, asynchronous processes. An HOS specification includes the information necessary to execute software with the maximum degree of simultaneous operation. Using HOS, it is possible to maintain complete asynchronous control of the processing hardware while minimizing storage use through dynamic memory allocation.

As discussed in Chapter 1, the primary goal of implementing a multiprocessing system is to gain the time (and therefore cost) advantage of concurrent computation. This chapter investigates the reasons that HOS makes possible the maximum use of all active processing hardware in an implementation and the minimization of passive storage use in the system during execution. The first section develops these concepts in comparison with other methods of software definition, and the second examines their form and implementation in the actual software of the Higher Order Machine.

4.1 Implementation Concepts for Multiprocessing Software

During the execution of concurrent processes, it is essential to determine when an instruction may be executed. This is so because it is desirable to execute all instructions as soon as possible and thereby maximize the system throughput. Equally important is the need to delay the allocation of storage space to a particular datum until its value is generated and to release that storage for further service as soon as its value is no longer needed. Through this practice, the use of the available space may be intensified to as great a degree as possible. This will minimize the overall memory requirements for the system and maximize its cost-effectiveness. This section investigates the manner in which these goals may be attained in the implementation of a multiprocessing system through proper specification of software. The implementation concepts that are developed are the determination of the execution readiness of an instruction, the unique identification of each entity of data within the system, and the scope limitations of data access for software modules.

4.1.1 Instruction Readiness

Functionally, any primitive hardware operation in a system is ready to begin execution as soon as the values of all its input operands have been defined and are available. This is the only necessary criterion. In an HOS specification, this condition is detectable dynamically because of adherence to the single-assignment property. This has been described for single-assignment programming in Section 3.2.3. This information is much the same as that provided by a data-flow layout in which each datum, or token, is uniquely identified not by name, but by its location on a particular link. In data-flow layouts, separate paths are provided for each datum so that an operation can identify each input uniquely as it becomes available on the link. Single-assignment programming techniques provide for unique identification of each datum and for flags at each instruction to signal the availability of each input operand. (Of course, the use of some

flag mechanism is actually the only way that a data-flow layout, as well, could be implemented unless it were hardwired.)

This specification of instruction readiness does not exist in sequential programming. Each datum is identified by its storage-space address and the same memory location is used over and over for different data, so, as a result, the execution of each operation must be made in proper sequence under the assumption that all its input operands are properly defined. This condition must be guaranteed in the program design, and thus creates, to a large extent, the need for programming skill in the software development as well as causing greater propensity for program anomalies.

4.1.2 Data Entity Concept

If the single-assignment technique is to benefit a multiprocessing system that must have cost-effective memory consumption, then the concept of an autonomous, uniquely identified datum must be introduced. It is necessary that the identity of each entity of data be made independent of the manner in which its value is stored. Only in this way can the same memory location be reused for different data while retaining the explicit definition of execution readiness for each operation. This storage independence is lacking in single-assignment programming and results in economically unfeasible memory consumption.

Once the software specification is made independent of the characteristics of the machine, then more explicitness can be required of the specification, thus eliminating any implied conditions or restraints. An HOS specification includes this definitional exactness. Specifically, to make possible the explicit definition of operation readiness as well as reuse of storage locations, it is necessary to require that a variable be referenced only once as well as assigned only once. This single-

reference property requires that a new primitive operation be defined which would perform this replication function. This operation, called a CLONE in AXES, would input a single datum and output a datum set, each member having a unique identity but also having a value that is the same as that of the input. This function is analogous to the duplication branch in a data-flow definition (called a "wye" operation).

In implementation, it is not necessary that physical copies be made of a datum that is referenced more than once. The single-reference constraint and the replication function are required for definitional explicitness, not for machine operation. It is through this practice that it is possible to determine dynamically when all references have been made to a datum so that the memory space it has been occupying can be reallocated. Since all references are specified and each is uniquely identified, a counting mechanism may be maintained which will indicate when a value can be discarded. However, in a multiprocessing environment, it may prove more time-saving actually to create a new physical copy for each reference. This would eliminate memory contention which can consume an inordinate amount of processor time. For simple data types, this practice may not use much more memory space because nearly as much space is required to store an address reference to a value as to store the value itself.

4.1.3 Software Module Scope

The single reference and single assignment of variables in the specification of software have been discussed in the previous sections as definitional constraints which permit dynamic memory allocation and the detection of instruction readiness. In HOS, these requirements result from axioms which govern the scope limitations of data within a software module. An HOS specification is a hierarchy of software modules which must be constructed in a manner consistent with the specification

axioms. (The consequence of compliance with these axioms at the design level of abstraction with respect to interface correctness and data-type operations are developed in [HAM76a,b,c].) With respect to data, an HOS module is a unified functional mapping from the module input to the module output. The module input and output are different data sets and are the only means through which a module may have external communication. Side effects due to data can therefore not exist in a software operation. (Side effects due to relative timing and undefined functional mappings are prevented by restrictions resulting from other axioms on the ordering of submodules and the rejection of invalid inputs). The data scope constraints imposed by the HOS axioms can be summarized as follows:

a. All data identified within a module are local to the module except members of the module's input and output sets.

b. A module may not assign values to any data external to itself unless that data is part of its output set.

c. An invoked module has reference access only to external data that is part of its input set.

d. A module may reference but not assign the values of its input set. (This results in the single-assignment constraint).

e. A controlling module is solely responsible for providing the input set and receiving the output set of a module which it invokes. Therefore, no other module has access to the data within an invoked module except the controller which makes the invocation, and this has access to only the input and output sets.

These data-access constraints permit the single-reference requirement in an HOS specification, whereas the lack of scope limitations in a single-assignment program conceals data concordance. This makes it impossible during the execution of a single-assignment program to determine when the references to a datum have all been made.

4.2 Instruction Readiness, Memory Allocation, and Module Scope Conclusions

The results of this discussion may be summarized by attributing the ability to detect instruction readiness to the single assignment of variables, the capability of performing dynamic memory allocation to the single reference of variables, and the prevention of data-induced side-effects to module-scope constraints. An instruction may be executed when all its input operands have been defined. Under single-assignment, this condition exists as soon as all the input variables have been assigned values. In conjunction with this, the single-reference property permits the determination of the dynamic allocation of memory to a variable--memory is allocated when its value is assigned and is released after its value has been referenced (or after all references have been made if a counting mechanism is used instead of separate copies for each reference). At the module level, these two constraints cause a module to be functionally equivalent to an operation with deterministic behavior and no data side-effects, i.e., the module references only its input and it assigns its output through a deterministic functional mapping. The implementation of these concepts within the Higher Order Machine has been outlined.

5.0 CONCLUSION

The Higher Order Machine concept has been presented as a solution to the problem of utilizing the full computational power of currently available hardware. In comparison with a cross-section of extant multiprocessing architectures, the Higher Order Machine (HOM) has been found to be uniquely capable of:

1. any degree of processing concurrency thereby maximizing computational throughput,
2. total dynamic memory allocation which minimizes storage consumption,
3. fully-distributed operation which provides maximum hardware utilization (both passive and active) and insures greatest cost-effectiveness,
4. a complete modularity in construction which permits the hardware configuration to be determined by the application requirements,
5. the reconfiguration of Primitive Hardware Operations which allows optimization of the hardware/software implementation trade-off,
6. the simplicity and reliability that results from the elimination of the excess hardware required for control of systems that are not structured by HOS.

These hardware capabilities become possible only through the formalized specification of software according to the principles of Higher Order Software (HOS). An HOS specification is an abstract hierarchical decomposition depicting the functional characteristics of a system, rather than a set of operations on fixed hardware. It therefore defines interactions of data at any level of abstraction and permits the automatic identification of both Primitive Hardware Operation readiness and dynamic memory allocation which together produce the unique capabilities of the HOM. The modularity and distributed control inherent in HOS and reflected in the structure of the HOM facilities the incorporation of fault-tolerant capabilities throughout the system. This allows the design of safety-critical systems to have a greater dependence on the reliable operation of computational equipment.

The ability of the HOM to provide unlimited processing concurrency and minimized storage consumption with sufficient reliability could make possible many applications for automatic computation that were previously not feasible. Of increasing importance in the development of complex systems is computer simulation to verify the system operation before great expense is invested in building operating prototypes. The HOM can provide fully-digital simulations that execute faster than real time without the expense of long hours of operation on costly high-speed hardware. Because increased accuracy and greater simulation complexity need not affect simulation execution time on the HOM, much more reliable results can be attained.

The organized use of extensive processing concurrency also can have great impact on the feasibility of automated manipulation and control systems. The development of automated construction and assembly facilities has been hindered by the volume of data that must be processed simultaneously in order that the system operate in real time. More accurate, flexible control systems can thus be made possible with concurrent processing, relieving the time constraints on control-law computations for time-critical tasks. When this capability is provided with the reliability necessary for safety-critical applications, the performance of life-critical functions such as in tactical-communications networks and aircraft-flight control can be greatly enhanced by increased reliance on automatic systems. The key to these and many other improved-performance applications is the availability of sufficient concurrent computation with sufficient reliability for the minimum necessary investment in cost, weight, and size of processing hardware. This is what can be provided by the HOM for systems specified by HOS.

REFERENCES

CHA71 Chamberlin, D.D. "The 'Single-Assignment' Approach to Parallel Processing."

HAM76a Hamilton, M. and Zeldin, S. "Higher Order Software--A Methodology for Defining Software." IEEE Transactions in Software Engineering, Vol. SE-2, No. 1, March 1976.

HAM76b Hamilton, M. and Zeldin, S. "Integrated Software Development System/Higher Order Software Conceptual Description," Version 1. Cambridge, MA: Higher Order Software, Inc. November 1976.

HAM76c Hamilton, M. and Zeldin, S. "AXES Syntax Description." Cambridge, MA: Higher Order Software, Inc. December 1976.

RUM75 Rumbaugh, J. "A Parallel Asynchronous Computer Architecture for Data Flow Programs." Project MAC, Cambridge, MA: Massachusetts Institute of Technology. May 1975.

THU75 Thurber, K.J. and Wald, L.D. "Associative and Parellel Processors." ACM Computer Surveys, Vol. 7, No. 4, December 1975.

SMI73 Smith, III, T.B. "A Highly Modular Fault-Tolerant Computer System." Document T-595. Cambridge, MA: The Charles Stark Draper Laboratory, Inc., November 1973.

SYL75 Sylvain, P., et al. "The Design and Evaluation of the Array Machine: A High-Level Language Processor." Computer Science Department, University of CA, Los Angeles, CA, 1975.